SECURING EMBEDDED NETWORKS THROUGH SECURE COLLECTIVE ATTESTATION

Vom Fachbereich Informatik (FB 20)
an der Technischen Universität Darmstadt
zur Erlangung des akademischen Grades eines Doktor-Ingenieurs
genehmigte Dissertation von:

M.Sc. Ahmad Ibrahim
Geboren am 27. July 1989 in Bent Jbeil, Lebanon

Referenten:
Prof. Ahmad-Reza Sadeghi (Erstreferent)
Prof. Gene Tsudik (Zweitreferent)

Tag der Einreichung: 06. February 2019
Tag der Disputation: 18. March 2019

TECHNISCHE
UNIVERSITÄT
DARMSTADT

System Security Lab
Intel Collaborative Research Center for Collaborative Autonomous
& Resilient Systems
Fachbereich für Informatik
Technische Universität Darmstadt

Hochschulkennziffer: D17

Ahmad Ibrahim:
Securing Embedded Networks through Secure Collective Attestation
Darmstadt, Technische Universität Darmstadt
Jahr der Veröffentlichung der Dissertation auf TUprints: 2019
URN: urn:nbn:de:tuda-tuprints-85883
Tag der mündlichen Prüfung: 18.03.2019

ABSTRACT

Networks of embedded devices are becoming increasingly popular. Examples of such networks range from small ecosystems, such as home and building automation, to very large infrastructure, e.g., industrial control systems. Devices in these networks usually collect private information and perform safety- and security-critical operations. Therefore, attacks targeting them are critical as they threaten both privacy and safety of humans, and are capable of causing extreme physical damage. A prominent example of such attacks is characterized by the Stuxnet worm which targets industrial control systems and is suspected to have caused substantial damage to Iran's nuclear program. In fact, three classes of attacks are relevant in the context of large embedded networks. These are malware infestation, physical, and runtime attacks.

In this dissertation, we investigate the security of large embedded networks in different deployment scenarios and provide security solutions that allow to scalably secure and manage these networks. In particular, we identify the adversarial assumptions and security requirements for every scenario and provide security protocols, based on remote attestation, that allow the detection of attacks belonging to the three aforementioned classes.

In order to secure large embedded networks, this dissertation presents the design and implementation of several scalable attestation protocols for centralized and autonomous networks. First, we present three scalable attestation protocols for centralized embedded networks that allows the detection of malware infestation attacks. These are accompanied with a systematic treatment of the problem that allows identifying and fulfilling all security requirements. Second, we investigate the problem of physical attacks on large embedded networks defining the capabilities of a physical attacker, and design two scalable attestation protocols that efficiently detect physical attacks in both centralized and autonomous settings. Third, we design a scalable attestation protocol that is capable of efficiently detecting runtime attacks on autonomous systems without disrupting the functionality or safety requirements of these systems. Finally, since management and software updates represent a critical requirement for securing a device as well as an important entry point for attackers, we also present a scalable management protocol for large networks that allows to securely and efficiently broadcast management commands and collect statistics regarding all devices in an embedded network.

ZUSAMMENFASSUNG

Netzwerke eingebetteter Geräte werden immer beliebter. Beispiele solcher Netzwerke reichen von relativ kleinen Umgebungen wie der Haus- und Gebäudeautomation bis hin zu sehr großen Infrastrukturen, z. B. industriellen Steuerungssystemen. Geräte in diesen Netzwerken sammeln oft private Informationen und führen sicherheitskritische Vorgänge aus. Daher sind Angriffe, die auf sie abzielen, von entscheidender Bedeutung, da sie sowohl die Privatsphäre als auch die Sicherheit von Personen bedrohen und zu großen Schäden führen können. Ein prominentes Beispiel für solche Angriffe ist der Stuxnet-Wurm, der auf industrielle Kontrollsysteme abzielt und vermutlich das iranische Nuklearprogramm erheblich beschädigt hat. Tatsächlich sind insbesondere drei Angriffskategorien im Zusammenhang mit großen eingebetteten Netzwerken relevant. Diese sind Malware-Befall, physische Angriffe sowie Laufzeitangriffe.

In dieser Dissertation untersuchen wir die Sicherheit großer eingebetteter Netzwerke in verschiedenen Implementierungsszenarien und stellen Sicherheitslösungen vor, mit denen diese Netzwerke auf eine skalierbare Weise gesichert und verwaltet werden können. Insbesondere identifizieren wir relevante Annahmen über den Angreifer und jeweilige Sicherheitsanforderungen für jedes dieser Szenarien und stellen auf Remote-Attestierung beruhende Sicherheitsprotokolle vor, mit denen Angriffe der drei zuvor genannten Klassen erkannt werden können.

Um große eingebettete Netzwerke abzusichern, werden in dieser Dissertation mehrere skalierbare Attestierungsprotokolle für zentralisierte und autonome Netzwerke entworfen und implementiert. Zunächst stellen wir drei skalierbare Attestierungsprotokolle für zentralisierte eingebettete Netzwerke vor, die die Erkennung von Malware-Angriffen ermöglichen. Diese werden begleitet von einer systematischen Behandlung des Problems, durch die alle Sicherheitsanforderungen erkannt und erfüllt werden können. Zweitens untersuchen wir das Problem physischer Angriffe auf große eingebettete Netzwerke, definieren die Fähigkeiten eines physischen Angreifers und entwerfen zwei skalierbare Attestierungsprotokolle, mit denen physische Angriffe sowohl in zentralen als auch in autonomen Umgebungen effizient erkannt werden können. Drittens entwerfen wir ein skalierbares Attestierungsprotokoll, welches Laufzeitangriffe auf autonome Systeme effizient erkennen kann, ohne die Funktionalität oder Sicherheitsanforderungen dieser Systeme zu beeinträchtigen. Da Management- und Softwareupdates eine wichtige Voraussetzung fÃŒr die Sicherung eines Gerätes sowie einen wichtigen Einstiegspunkt fÃŒr Angreifer darstellen, stellen wir außerdem ein skalierbares Verwaltungsprotokoll für große Netzwerke vor, mit dem Verwaltungsbefehle sicher und effizient gesendet und Statistiken über allen Geräten in einem eingebetteten Netzwerk erfasst werden können.

ACKNOWLEDGMENTS

I would like to thank my first supervisor Prof. Ahmad-Reza Sadeghi for the opportunity to work at the Technical University of Darmstadt.

I would also like to thank my second supervisor Prof. Gene Tsudik for his help and support, as well as fruitful discussions and useful feedback.

I would like to thank the rest of my thesis commission: Prof. Stefan Katzenbeisser, Prof. Thomas Schneider, and Prof. Sebastian Faust, for their hard questions and insightful comments.

I am grateful to Dr. Matthias Schunter and Dr. Christian Wachsmann for their supervision, support, and positive feedback.

I am also grateful to Moreno Ambrosin, Prof. Mauro Conti, and Dr. Gregory Neven for the fruitful discussions and successful collaborations.

A very special gratitude goes out to my colleagues at the Technical University of Darmstadt, with a special mention to Shaza Zeitouni, Ghada Dessouky, Raad Bahmani, Tigist Abera, and Markus Miettinen.

Thank you for your collaboration, feedback, and of course friendship!

And most importantly, I am grateful to my family and friends who have provided me with emotional and moral support along the way.

Thanks for all your encouragement!

CONTENTS

1 INTRODUCTION 1
1.1 Goal and Scope of this Dissertation . . . 3
1.2 Summary of Contributions . . . 4
1.3 Outline . . . 5
1.4 Previous Publications . . . 6
2 PRELIMINARIES 9
2.1 Notation . . . 9
2.2 Mathematic and Cryptographic Background . . . 9
2.2.1 Computational Intractability Assumptions . . . 10
2.3 Cryptographic Primitives . . . 10
2.3.1 Hash Functions . . . 10
2.3.2 Encryption Schemes . . . 11
2.3.3 Authentication Schemes . . . 12
3 RELATED WORK 15
3.1 Attestation Landscape . . . 15
3.1.1 Taxonomy of Attestation . . . 16
3.1.2 Taxonomy of Attacks . . . 17
3.2 Requirements for Attestation . . . 20
3.2.1 Hardware Requirements . . . 20
3.2.2 Software Requirements . . . 21
3.2.3 Cryptographic Requirements . . . 21
3.3 Properties of Attestation . . . 22
3.3.1 Performance Properties . . . 22
3.3.2 Security Properties . . . 23
3.4 Software-Based Attestation . . . 24
3.4.1 Time-Based Attestation . . . 24
3.4.2 Untimed Software-Based Attestation . . . 26
3.5 Hardware-based Attestation . . . 27
3.5.1 TPM and TPM-based Attestation . . . 27
3.5.2 Others . . . 30
3.6 Hybrid Attestation . . . 31
3.6.1 Minimalist Approach for Attestation . . . 31
3.6.2 Attestation based on Isolation . . . 32
3.7 Runtime Attestation . . . 33
3.7.1 Dynamic Attestation . . . 33
3.7.2 Control-Flow Attestation (CFA) . . . 34
3.8 Comparison . . . 35
3.9 Conclusion and Open Problems . . . 37
4 DETECTION OF MALWARE INFESTATION 39
4.1 Scalable Embedded Device Attestation . . . 39

4.1.1 Collective Attestation . . . 40
4.1.2 Protocol Description . . . 43
4.1.3 Implementation . . . 48
4.1.4 Performance Evaluation . . . 49
4.1.5 Security Analysis . . . 54
4.1.6 Protocol Extensions . . . 55
4.1.7 Conclusion . . . 57
4.2 Secure and Scalable Aggregate Network Attestation . . . 58
4.2.1 Collective Attestation . . . 59
4.2.2 Proposed Signature Scheme . . . 62
4.2.3 Protocol Description . . . 66
4.2.4 Implementation . . . 69
4.2.5 Performance Evaluation . . . 70
4.2.6 Security Analysis . . . 75
4.2.7 Threshold Attestation . . . 77
4.2.8 Conclusion . . . 78
4.3 Systematic Treatment of Collective Attestation . . . 79
4.3.1 Definitions . . . 80
4.3.2 Properties of Collective Attestation . . . 84
4.3.3 Features of Collective Attestation . . . 86
4.3.4 Protocol Analysis . . . 88
4.3.5 Generic Solution . . . 90
4.3.6 Implementation . . . 92
4.3.7 Performance Evaluation . . . 93
4.3.8 Conclusion . . . 94
4.4 Related Work . . . 95
4.5 Conclusion . . . 96
5 DETECTION OF PHYSICAL ATTACKS 97
5.1 Device Attestation Resilient to Physical Attacks . . . 97
5.1.1 Collective Attestation . . . 98
5.1.2 Protocol Description . . . 101
5.1.3 Implementation . . . 109
5.1.4 Performance Evaluation . . . 111
5.1.5 Security Analysis . . . 114
5.1.6 Conclusion . . . 115
5.2 Unattended Scalable Attestation of Embedded Devices . . . 116
5.2.1 Collective Attestation . . . 117
5.2.2 Protocol Description . . . 121
5.2.3 Implementation . . . 128
5.2.4 Performance Evaluation . . . 130
5.2.5 Security Analysis . . . 134
5.2.6 Protocol Extensions . . . 136
5.2.7 Conclusion . . . 138
5.3 Related Work . . . 138

5.4 Conclusion . . . 139
6 DETECTION OF RUNTIME ATTACKS 141
6.1 Collective Attestation . . . 142
6.1.1 Problem Description and System Model . . . 142
6.1.2 Requirements Analysis . . . 142
6.2 Control-flow Attestation . . . 144
6.2.1 Multiset Hash Function . . . 145
6.2.2 Efficient CFA . . . 145
6.3 Protocol Description . . . 148
6.3.1 Interaction of Two Devices . . . 148
6.3.2 Interaction of Multiple Devices . . . 149
6.4 Implementation and Evaluation . . . 149
6.4.1 Implementation . . . 150
6.4.2 Evaluation . . . 150
6.4.3 Network Simulation . . . 151
6.5 Security Analysis . . . 153
6.6 Conclusion . . . 155
7 SECURE AND SCALABLE MANAGEMENT 157
7.1 Scalable Management . . . 158
7.1.1 Problem Description and System Model . . . 158
7.1.2 Requirements Analysis . . . 159
7.1.3 Management Finite State Machine . . . 161
7.2 Secure Data Aggregation . . . 163
7.3 Protocol Description . . . 164
7.3.1 Command Distribution Protocol . . . 164
7.3.2 Statistics Collection protocols . . . 166
7.4 Implementation . . . 167
7.5 Performance Evaluation . . . 169
7.6 Security Analysis . . . 172
7.7 Related Work . . . 173
7.8 Conclusion . . . 174
8 DISCUSSION AND CONCLUSION 177
8.1 Dissertation Summary . . . 177
8.2 Future Research Directions . . . 178
9 ABOUT THE AUTHOR 181

BIBLIOGRAPHY 185

LIST OF FIGURES

Figure 3.1 Taxonomy for attestation . . . 16
Figure 3.2 Taxonomy for attacks . . . 18
Figure 4.1 Example 8-device network: $D_1, \ldots, D_8$ . . . 42
Figure 4.2 Protocol attdev . . . 45
Figure 4.3 Protocol attest . . . 46
Figure 4.4 Implementation based on SMART [52] . . . 48
Figure 4.5 Implementation based on TrustLite [84] . . . 49
Figure 4.6 Performance of attestation per device . . . 51
Figure 4.7 Performance of attestation for tree topologies . . . 52
Figure 4.8 Performance of attestation for chain and star topologies . . . 52
Figure 4.9 Performance of attestation for networks with tree topologies and varying numbers of neighbors per device . . . 53
Figure 4.10 Performance of our attestation compared to the naïve approach . 53
Figure 4.11 Comparison between interactive and non-interactive collective attestation . . . 54
Figure 4.12 Example 7-device network (four aggregators and five provers) . . 59
Figure 4.13 Protocol tokenReq . . . 67
Figure 4.14 Protocol attest . . . 68
Figure 4.15 Implementation based on SMART [52] . . . 70
Figure 4.16 Implementation based on TrustLite [84] . . . 71
Figure 4.17 Performance of attestation . . . 74
Figure 4.18 Comparison to the other solution . . . 74
Figure 4.19 Performance of attestation as function of the number of malicious provers . . . 75
Figure 4.20 Verification of threshold attestation using an EC2 t2.micro verifier 78
Figure 4.21 Example network of nine members . . . 80
Figure 4.22 Protocol CA . . . 91
Figure 4.23 Implementation based on SMART [52] . . . 92
Figure 4.24 Implementation based on TrustLite [84] . . . 93
Figure 4.25 Runtime of the generic solution . . . 94
Figure 5.1 protocol beat (as viewed by D_i) . . . 103
Figure 5.2 protocol collect (as viewed by V) . . . 104
Figure 5.3 protocol collect (as viewed by D_i) . . . 105
Figure 5.4 Protocol attest . . . 107
Figure 5.5 Implementation based on SMART [52] . . . 109
Figure 5.6 Implementation based on TrustLite [84] . . . 110
Figure 5.7 Energy consumption per device . . . 112
Figure 5.8 protocol beat . . . 113
Figure 5.9 protocol collect/ attest . . . 113

Figure 5.10 Example 8-device network: $D_1, \ldots, D_8$ . . . 120
Figure 5.11 Protocol join . . . 123
Figure 5.12 Protocol attest . . . 124
Figure 5.13 Merkle Hash Tree (MHT) of software configurations . . . 125
Figure 5.14 Protocol beat . . . 126
Figure 5.15 checkTime **on** D_j . . . 126
Figure 5.16 Protocol heal . . . 127
Figure 5.17 Implementation based on SMART [52] . . . 129
Figure 5.19 authenticate **on** D_i . . . 129
Figure 5.20 verify **on** D_i . . . 129
Figure 5.18 Implementation based on TrustLite [84] . . . 130
Figure 5.21 Energy consumption per device . . . 131
Figure 5.22 Energy consumption of beat . . . 132
Figure 5.23 Performance of all protocols . . . 132
Figure 5.24 Performance in terms of network size . . . 133
Figure 5.25 Comparison to other solutions based on TrustLite [84] . . . 133
Figure 5.26 Performance of heal on TrustLite [84] . . . 134
Figure 5.27 checkTime **on** D_j . . . 138
Figure 6.1 Example 5-device network: $D_1, \ldots, D_5$ . . . 144
Figure 6.2 Control-Flow Graph (CFG) showing C-FLAT and our CFA scheme 146
Figure 6.3 Simple explanatory CFGs . . . 147
Figure 6.4 Protocol interact . . . 148
Figure 6.5 Simulated collaboration scenarios (OMNeT++ [104]) . . . 151
Figure 6.6 Performance of our solution in serial and parallel collaboration scenarios . . . 152
Figure 6.7 Performance of our solution in hybrid collaboration scenario . . . 152
Figure 7.1 System model as a network of devices; each device acts as at least one of the following entities: endpoint (v_j), aggregator (a_l), and cache (c_k) . . . 159
Figure 7.2 A simple sub-M-FSM with 3 states and 4 transitions. One of the transitions is to the starting state of a different sub-M-FSM . . . 162
Figure 7.3 Example: Software update management . . . 162
Figure 7.4 Command distribution protocol based on µTesla . . . 165
Figure 7.5 Client Agent Module of managees for Riot-OS . . . 168
Figure 7.6 Performance of command distribution . . . 170
Figure 7.7 Performance of statistics collection . . . 171

LIST OF TABLES

Table 3.1 Comparison between different classes of attestation . . . 36

Table 4.1 Performance of cryptographic functions 73
Table 4.2 Performance of OAS algorithms 73
Table 5.1 Runtime of `Primitives` 131
Table 7.1 Performance of cryptographic functions 170

1 INTRODUCTION

Embedded systems are increasingly pervading every aspect of our daily lives. The number of deployed systems has been rapidly growing in the past decade. According to Cisco, 50 billion of such systems are expected to be deployed by 2020. The ever increasing development and deployment of embedded systems, in addition to their interconnection through the legacy internet has enabled a wide range of applications. This phenomenon is referred to as the Internet of Things (IoT). Examples of IoT deployment include personal devices such as smart gadgets, small ecosystems such as home/office automation, and very large scale deployments such as industrial control systems and autonomous systems.

Embedded system are usually special-purpose devices designed to perform restricted tasks. They have constrained resources in terms of memory, computational power, and energy. They are also constrained in terms of size. As a consequence, such systems lack the security features of legacy computing device such as memory virtualization, and secure cryptographic co-processor, e.g., Trusted Platform Module (TPM). Moreover, due to their prevasiveness, increasing sensing and actuating capabilities, and their processing of sensitive information, embedded systems represent an attractive target for attacks. They have been the target of several successful attacks that actually caused real physical damage. Popular examples include the Stuxnet episde [148], and the recent HVAC [149] and Jeeb attacks [5].

Attacks on embedded systems range from remote software attacks, where the attacker manipulates a device's software state from afar by either installing malicious code also known as malware infestation or by mounting runtime attacks that alter the execution of the code without changing its binaries, to physical or hardware attacks where the attacker exploits physical proximity to a victim device to modify its hardware and/or extract its secrets. Unlike physical attacks, remote software attacks are scalable as they require no physical proximity to the victim device, whereas the attacker is limited to physically attacking a small number of devices.

The most prominent example of malware infestation is the stuxnet worm [148] that targeted nuclear centrifugers leading to catastrophic consequences. Similarly, the power of runtime attacks has been demonstrated by the turing complete Return-Oriented Programming (ROP) attack [127], where the attacker exploits a memory corruption vulnerability such as a buffer overflow to maliciously modify control-flow data and execute an arbitrary malicious functionality. Finally, physical attacks can be either invasive requiring the adversary to disassemble the device and extract its components, or non-invasive such as side channel attacks that exploit physical properties of the device, e.g., electromagnetic radiations, to extract device's secrets during normal operation. Physical attacks on stand-alone devices such as personal gadgets are insignificant, as such devices are almost always attended. However, devices in very large installation are spread over a

very large area and can be within the attacker's grasp, e.g., equipment outside a smart factory or drones in an autonomous system.

Remote attestation is a security service that allows a trusted entity denoted by *verifier* to assess the trustworthiness of a remote (possibly malicious) device – the *prover*. It is established as an interactive protocol through which the prover sends the verifier a measurement of its software state. The measurement is typically a cryptographic hash of the prover's binary. It serves as a proof that the prover's software has not been maliciously modified, i.e., has not been infested by malware. This is referred to as static attestation. Several static attestation schemes have been proposed in the literature. These can be clustered into three main classes: Software-based attestation [81, 126, 125, 122, 62, 89], hardware-based attestation [111, 144, 86, 118, 97, 95], and hybrid attestation [52, 84, 57, 28].

Software-based attestation is applicable to legacy and low-end devices as it requires neither secure hardware nor cryptographic secrets. Its security is rather based on the limited computational power of the prover and a strict estimation of the time required to generate a measurement of its software state. The security of software-based attestation is based on strong assumptions that are hard to achieve in practice [15], such as the adversarial silence during the execution of the attestation protocol, and the optimality of measurement function and its implementation. For this reason software-based attestation is considered impractical and its applicability is limited to narrow range of scenarios, e.g., attestation of peripherals. Hardware-based attestation exploits cryptographic co-processors, such as TPM, to ensure the integrity and authenticity of the measurement. However, its hardware requirements is often too complex and too expensive for embedded systems. Hybrid attestation schemes aim at reducing the hardware requirements for securing remote attestation. Its is based on devising lightweight security architectures that provide similar securiy gaurantees to those of hardware-based attestation while minimizing the required hardware security features. Examples of such architecture include SMART [52] and TrustLite [84] that only require a Read-Only Memory (ROM) and a simple Memory Protection Unit (MPU).

Several recent efforts has lead to the development of Control-Flow Attestation (CFA) schemes. In these schemes the software state measured by the prover is not a hash of the software binaries. It is rather a compact representation of the precise control-flow path followed by the software during execution. Based on the reported measurement and a database of benign execution paths, the verifier can detect any deviation from the expected execution of the software, thus detecting control-flow attacks such as ROP [127]. CFA [9, 49, 155] induces a very large overhead on the prover, who is required interrupt and record every control-flow event, e.g., indirect branches, during software execution; and on the verifier, who is expected to store and search a very large database of benign execution paths in order to verify one reported measurement. The prover's overhead problem has been addressed by recent CFA schemes [49, 155] that leverage hardware assistance in order to record the control-flow of a software in parallel to its execution. These schemes allow CFA with zero runtime overhead on the prover. However, the verifier's overhead continues to limit applicability of CFA to very small programs and hinders its scalability to large networks such as autonomous systems.

While current industrial trends envision networks of embedded systems that consist of a very large number of heterogeneous mobile devices, all existing attestation schemes are geared to the single-device setting and are neither scalable nor directly applicable to such envisioned networks. For example in industrial control systems a very large number of devices collaborate to monitor safety-critical processes, and in autonomous networks intelligent devices collaborate to collectively achieve a common task. On the other hand, static attestation schemes are not scalable, i.e., they cannot efficiently detect malware infestation for a large number of devices, and CFA schemes have high verification overhead and are only applicable to small-size programs. Moreover, all these schemes vulnerable to physical attacks that are considered out of scope of single-device attestation schemes while are in fact extremely relevant in emerging scenarios.

1.1 GOAL AND SCOPE OF THIS DISSERTATION

We aim at securing emerging networks of embedded systems by providing security solutions that allow scalable integrity verification as well as secure management and monitoring of such networks. We extend traditional remote attestation to provide efficiency, scalability, and detection of stronger attackers that either have physical proximity to the devices or are capable of performing runtime attacks. In particular, the main goal of this dissertation is introducing the design and implementation of security solutions that are capable of efficiently detecting a wide range of attacks on embedded systems that are relevant in large deployments – collective attestation, as well as enabling secure management and monitoring of embedded systems in such deployments. In general, we focus on designing collective attestation schemes that are capable of detecting malware infestation, runtime attacks, and physical attacks. Our solutions targeting malware infestation also take into account the possibility of physical attacks in the targeted settings as well as the severity of Denial of Service (DoS) attacks. Our runtime attack detection exploits software modularity and devises a novel execution path representation to allow efficient CFA of complex embedded software. And, our physical attack detection assumes that in order to physically attack a device, the attacker has to turn it off for a non-negligible amount of time. Thus it is capable of detecting invasive physical attacks that require the attacker to disassemble the device and extract its components. However, non-invasive attacks such as side channel attacks are considered out of scope. Finally, we provide a secure management scheme that leverage cache-capable networks and aggregation trees of untrusted nodes to allow secure management and assessment of embedded devices using a single low-power management device, e.g., a smart phone.

It is important to note that networks of embedded devices can be either (1) centralized involving a central entity that is responsible running and monitoring the network, e.g., IoT devices in smart environments or industrial control systems in a smart factory; or (2) decentralized where a multitude of devices perform a collaborative task in distributed manner forming collectively intelligent systems, e.g., autonomous networks of drones or robots. In this dissertation we aim at securing both centralized and decentralized autonomous systems. In particular, we devise collective attestation schemes that are capable of detecting malware infestation and physical attacks in centralized networks

and autonomous networks. Furthermore, we devise a Control-Flow Attestation (CFA) scheme that is capable of securing collaboration between devices in autonomous networks. Finally, our secure management and assessment scheme requires a central management entity for distribution of management commands and collection of statistical information regarding managed devices.

1.2 SUMMARY OF CONTRIBUTIONS

The main contributions of this dissertation are listed below:

Systematization of Knowledge on Attestation. We survey the state of the art on attestation and systematically study and classify them. In particular, we present different possible attacks that are relevant to attestation, i.e., those attacks that either target or are detected by existing attestation schemes. We also describe different security and performance requirements imposed by different attestation schemes. Existing attestation schemes are then clustered based on their requirements and the attacks they mitigate and/or are vulnerable to.

Efficient Collective Attestation Solution. We solution and implement the first attestation solution for detecting malware infestation in large-scale systems. This scheme presents the first step in a new line of research on multi-device attestation. We further develop the first security model for collective attestation and show the security of our scheme in this model. Our approach demonstrates the feasibility of collective attestation as it enables attesting a million-device network in order of seconds. Efficiency is achieved by distributing the attestation burden across the network and using symmetric key cryptography for in-network interactions.

Secure Aggregation and Resiliency to Physical and DoS Attacks. We develop a novel aggregate signature – Optimistic Aggregate Signature (OAS), which is a combination of aggregate and multisignatures providing constant verification overhead while allowing aggregation of signatures on different messages. Based on OAS we present a collective attestation solution that is resilient against physical attacks on devices involved in the attestation. The developed solution also provides resiliency against Denial of Service (DoS) attacks on attestation by requiring the verifier to possess a secure token in order to perform attestation.

Systematic Treatment of Remote Attestation. We establish collective attestation in centralized networks on solid ground, by performing a careful analysis of its requirements, and systematically identifying its software, hardware, as well as protocol components. To this end, we extract the security properties that should be satisfied by collective attestation, and derive a minimal set of features that enables satisfying these properties. We further present practical implementations that provide these features.

Secure Detection of Physical Attacks. We devise a security solution that is capable of detecting both malware infestation and physical attacks on centralized networks of embedded devices. The solution is based on extending collective attestation with absence detection that allows devices to detect physical attacks on their peers and report

it to the verifier. The scheme is based on network-wide heartbeats (i.e., authenticated timestamps) that are periodically broadcasted by each device in the network proving its presence at the time indicated by the timestamp.

Collective Attestation for Autonomous Systems. We present a secure collective attestation solution for autonomous systems that is capable of detecting both malware infestation and physical attacks. Our solution combines continuous peer-to-peer attestation and periodic local heartbeats with key exchange in order to detect malicious devices in the system. It also allows devices to be mobile based on a dedicated roaming protocol, where mobile devices collect proofs of trustworthiness from their old neighbors and exploit them for establishing new neighbors.

Healing and Attestation for Autonomous Systems. We present the first attestation solution that allows disinfecting malicious devices by re-installing genuine benign software. The solution is based on using Merkle Hash Tree (MHT) for measurement of software state, which allows identifying the exact memory address(es) that were modified by the adversary. Based on this novel measurement scheme, we devise a healing protocol that allows efficient disinfection of malicious devices.

Secure Collaboration Based on Control-Flow Attestation (CFA). We develop a security solution that is capable of securing interaction between devices in autonomous networks based on efficient CFA. In particular, we present a novel execution path representation based on Multiset Hash (MSH) functions that enables efficient verification of execution paths. Moreover, we decompose the software running on autonomous devices into small interacting software modules. Based on data-flow monitoring, each software module that is relevant for the generation of every piece of data is identified. In order to secure devices' interaction, we enable MSH-based CFA for each module that is relevant to data exchanged between devices. The software decomposition reduces the amount of code that should be attested drastically. Along with MSH representation it allows CFA of autonomous systems where the software is too complex and embedded devices need to act as both verifiers and provers.

Secure Management and Assessment of Large IoT Deployments. We devise a secure and scalable management framework that allows efficient management and assessment of large IoT deployments with a single low-power management device. Our framework has low storage, communication, and computation overhead on both the managed and managing devices. It leverages trees of untrusted nodes to distribute domain independent management specifications represented by finite state machines, and to efficiently collect statistics about the status of devices based on a novel secure aggregation protocol.

1.3 OUTLINE

This dissertation is organized as follows: We introduce our notation in Chapter 2 and provide a systematic study of state of the art on attestation in Chapter 3. Then we introduce our collective attestation solutions that are capable of efficiently detecting malware infestation in centralized networks in Chapter 4, where we also provide a systematic

treatment of such solutions. Next, in Chapter 5 we describe security solutions that are capable of detecting both malware infestation and physical attacks in centralized and autonomous settings. In Chapter 6 we present our security solution that allows securing interaction between collaborating devices in autonomous networks based on CFA. Our management and assessment protocol for large IoT deployments is described in Chapter 7. Finally, we conclude this dissertation in Chapter 8.

1.4 PREVIOUS PUBLICATIONS

This dissertation is based on several previous publications of the author. These publication are listed below. Note that, the author of this work is the main author of all these publications. He contributed to the sheer amount of ideas, design, implementation, and evaluation. For a full list of publications of the author, please refer to Chapter 9.

Chapter 3: Related Work

Ghada Dessouky, Ahmad Ibrahim, Ahmad-Reza Sadeghi. SoK: The Hitchhiker's Guide to the Attestation. To be submitted to *Proceedings of the 41st IEEE Symposium on Security and Privacy*, S&P'20, 2020.

Chapter 4: Detection of Malware Infestation

N. Asokan, Ferdinand Brasser, Ahmad Ibrahim, Ahmad-Reza Sadeghi, Matthias Schunter, Gene Tsudik, Christian Wachsmann. SEDA: Scalable Embedded Device Attestation. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, CCS'15, 2015.

Moreno Ambrosin, Mauro Conti, Ahmad Ibrahim, Gregory Neven, Ahmad-Reza Sadeghi, Matthias Schunter. SANA: Secure and Scalable Aggregate Network Attestation. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security*, CCS'16, 2016.

Ahmad Ibrahim, Ahmad-Reza Sadeghi, Shaza Zeitouni. SeED: Secure Non-Interactive Attestation for Embedded Devices. In *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec'17, 2017.

Ivan De Oliveira Nunes, Ghada Dessouky, Ahmad Ibrahim, Norrathep Rattanavipanon, Ahmad-Reza Sadeghi, Gene Tsudik. Toward Systematic Design of Collective Attestation Protocols. In *Proceedings of the 39th IEEE International Conference on Distributed Computing Systems*, ICDCS'19, 2019.

Chapter 5: Detection of Physical Attacks

Ahmad Ibrahim, Ahmad-Reza Sadeghi, Shaza Zeitouni, Gene Tsudik. DARPA: Device Attestation Resilient to Physical Attacks. In *Proceedings of the 9th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec'16, 2016.

Ahmad Ibrahim, Ahmad-Reza Sadeghi, Gene Tsudik. US-AID: Unattended Scalable Attestation of IoT Devices. In *Proceedings of the 37th IEEE International Symposium on Reliable Distributed Systems*, SRDS'18, 2018.

Ahmad Ibrahim, Ahmad-Reza Sadeghi, Gene Tsudik. HEALED: HEaling & Attestation for Low-end Embedded Devices. In *Proceedings of the 23rd International Conference on Financial Cryptography and Data Security*, FC'19, 2019.

Chapter 6: Detection of Runtime Attacks

Tigist Abera, Raad Bahmani, Ferdinand Brasser, Ahmad Ibrahim, Ahmad-Reza Sadeghi, Matthias Schunter. DIAT: Data Integrity Attestation for Resilient Collaboration of Autonomous Systems. In *Proceedings of the 26nd Annual Network and Distributed System Security Symposium*, NDSS'19, 2019.

Chapter 7: Secure and Scalable Management

Moreno Ambrosin, Mauro Conti, Ahmad Ibrahim, Ahmad-Reza Sadeghi, Matthias Schunter. SCIoT: A Secure and sCalable end-to-end management framework for IoT Devices. In *Proceedings of the 23rd European Symposium on Research in Computer Security*, ESORICS'18, 2018.

2

PRELIMINARIES

2.1 NOTATION

Let M be a finite set, then $|M|$ denotes the number of elements in M. For an integer (or a bit-string) n, $|n|$ denotes the bit-length of n. We denote by $m \in_R M$ the assignment of a uniformly sampled element of set M to a variable m. Furthermore, $\{0,1\}^\ell$ denotes the set of all bit-strings having length ℓ. For an event E, we denote by $\Pr[E]$ the probability that E occurs. E can be the result of a security experiment for example. A probability $\epsilon(\ell)$ is *negligible* if, for every polynomial f, $\epsilon(\ell) \leqslant 1/f(\ell)$ for all sufficiently large $\ell \in \mathbb{N}$.

If Q is a probabilistic algorithm, then $b \leftarrow Q(a)$ means that Q assigns its output to variable b on input a. Q^R denotes an algorithm Q that arbitrarily interacts with algorithm R while it is executing. We denote by the term $\mathsf{prot}\left[Q : a_Q;\ R : a_R;\ * : a_{pub}\right] \rightarrow [Q : b_Q;\ R : b_R]$ an interactive protocol prot between two probabilistic algorithms Q and R, through which, Q (resp. R) gets private input a_Q (resp. a_R) and public input a_{pub}. During the operation of Q (resp. R), it interacts with R (resp. Q). As a result, Q (resp. R) outputs b_Q (resp. b_R).

A network $\mathcal{N}$ is a set of interconnected device. The number of devices in the network is denoted by n. Devices in the network may be either homogeneous or heterogeneous in terms of software and hardware configurations. A device is denoted by D_i. Further, the network could be either static or dynamic in terms of topology or membership. Neighboring devices refer to device that have a direct communication link to each other. We denote that entity the is responsible for maintenance of the network by operator O. An entity that is interested in verifying the software integrity of the network is called verifier V. And the device through which the verifier communicate to the network is called the initiator D_1.

2.2 MATHEMATIC AND CRYPTOGRAPHIC BACKGROUND

Definition 2.1 (Admissible Pairing)**.** *Consider the three multiplicative groups $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ of prime order $p \approx 2^\ell$, where $\ell \in \mathbb{N}$ is some defined security parameter. $\mathbb{G}_1$, $\mathbb{G}_2$, and $\mathbb{G}_T$ are written multiplicatively with the identity element being 1. A mapping $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ is called a pairing if it has the following three properties:*

- *Bilinear. It should hold that $\forall O, O' \in \mathbb{G}_1$ and $\forall P, P' \in \mathbb{G}_2$:*

$$e(O \cdot O', P \cdot P') = e(O, P) \cdot e(O, P') \cdot e(O', P) \cdot e(O', P').$$

- *Non-degenerate. It should hold that $\forall O \in \mathbb{G}_1^*$, $\exists P \in \mathbb{G}_2^*$ (respectively $\forall P \in \mathbb{G}_2^*$, $\exists O \in \mathbb{G}_1^*$) such that $e(O, P) \neq 1$.*

- *Computable. There exists a probabilistic polynomial time algorithm that computes* $e(O, P)$ $\forall (O, P) \in \mathbb{G}_1 \times \mathbb{G}_2$.

Let g_1 and g_2 be generators for $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively, we call the pairing e admissible if $e(g_1, g_2) = g_T$, where g_T is a generator for $\mathbb{G}_T$.

2.2.1 *Computational Intractability Assumptions*

Definition 2.2 (Computational co-Diffie-Helman (co-CDH) Assumption)**.** *Let* $\mathbb{G}_1$, $\mathbb{G}_2$, *and* $\mathbb{G}_T$ *be three groups with large prime order* p, *let* g_2 *and* g_2^a *be two members of* $\mathbb{G}_2$ *where* $a \in_R \mathbb{Z}$, *let* h *be a member of* $\mathbb{G}_1$, *and let* $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ *be an admissible pairing. The co-CDH assumption states that every probabilistic polynomial time adversary* $\mathcal{A}$ *has negligible advantage (in security parameter* $\ell \in \mathbb{Z}$*):*

$$Adv_{\mathcal{A}}^{co-CDH} = Pr\left[\mathcal{A}(g_2, g_2^a, h) = h^a\right].$$

Definition 2.3 (Decisional co-Diffie-Helman (co-DDH) Assumption)**.** *Let* $\mathbb{G}_1$, $\mathbb{G}_2$, *and* $\mathbb{G}_T$ *be three groups with large prime order* p, *let* g_2 *and* g_2^a *be two members of* $\mathbb{G}_2$ *where* $a \in_R \mathbb{Z}$, *let* h *and* h^a *be two members of* $\mathbb{G}_1$ *where* $b \in_R \mathbb{Z}$, *and let* $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ *be an admissible pairing. Finally, Let* Pk_e *denote the tuple* $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, h, g_2, e)$. *The co-DDH assumption in* $\mathbb{G}_1$ *states that every probabilistic polynomial time adversary* $\mathcal{A}$ *has negligible advantage (in security parameter* $\ell \in \mathbb{Z}$*):*

$$Adv_{\mathcal{A}}^{co-DDH} = \left|Pr\left[1 \leftarrow \mathcal{A}(Pk_e, g_2^a, h^a)\right] - Pr\left[1 \leftarrow \mathcal{A}(Pk_e, g_2^a, h^b)\right]\right|.$$

Definition 2.4 (Gap co-Diffie-Helman (co-GDH) Group)**.** *A group* $\mathbb{G}_1$ *of prime order* p *is a co-GDH group if in this group: co-CDH assumption holds while co-DDH assumption does not.*

2.3 CRYPTOGRAPHIC PRIMITIVES

2.3.1 *Hash Functions*

Definition 2.5 (Hash Functions)**.** *Let* $\alpha, \beta \in \mathbb{N}$. *A hash function* hash *maps bit strings of arbitrarily finite length* α *to bit strings of fixed length* β, *i.e.,* $\mathsf{hash} : \{0,1\}^\alpha \to \{0,1\}^\beta$

Definition 2.6 (Hash Family)**.** *A family of hash functions constitutes the set* $\left\{\mathsf{hash}_k : \{0,1\}^\alpha \to \{0,1\}^\beta | k \in \{0,1\}^\kappa\right\}$ *of hash functions indexed by* k, *where* $\alpha, \beta, \kappa \in \mathbb{N}$.

Definition 2.7 (Collision Resistance)**.** *A hash function* hash *is called collision resistant if it is computationally infeasible to find two distinct inputs values* $a_0 \neq a_1$ *such that* $\mathsf{hash}(a_0) = \mathsf{hash}(a_1)$, *i.e., it should hold that for every probabilistic polynomial time adversary* $\mathcal{A}$*:*

$$Pr\left[\mathsf{hash}(a_0) = \mathsf{hash}(a_1) | (a_0, a_1) \leftarrow \mathcal{A}^{\mathsf{hash}}\right] \leqslant \epsilon(\alpha, \beta).$$

where ϵ *is negligible in* α *and* β.

Definition 2.8 (Multiset Hash (MSH) Functions). *A multiset denotes a finite unordered group of elements where an element may appear more than once. Let M_1 and M_2 be multisets of elements from a countable set S of cardinality $2^m, m \in \mathbb{Z}$, and let $\alpha, \beta \in \mathbb{N}$. A Multiset Hash Function is a tuple of probabilistic polynomial time algorithms* $(H, +_H, =_H)$. *On input a multiset M with a finite number $n \in \{0,1\}^\alpha$ of elements, H outputs a string h with a fixed length β, i.e., $h \leftarrow H(M)$. On input of two hashes h_1 and h_2 of two multisets M_1 and M_2 (i.e., $h_1 = H(M_1)$ and $h_2 = H(M_2)$), $=_H$ indicates whether the two multisets M_1 and M_2 are equal, i.e., $H(M_1) =_H H(M_2)$ is true iff $M_1 = M_2$. h_1 and h_2 are then called equivalent. On input of two hashes h_1 and h_2 of two multisets M_1 and M_2, $+_H$ outputs the hash h_a of the addition of the two multisets M_1 and M_2, i.e., $H(M_1 \uplus M_2) =_H H(M_1) +_H H(M_2)$.*

Definition 2.9 (Multiset Collision Resistance). *Consider a MSH function as defined in Definition 2.8* $(H, +_H, =_H)$ *that has security parameters $\alpha, \beta \in \mathbb{N}$ and let S be a set of cardinality $2^m, m \in \mathbb{Z}$. This MSH function is called multiset-collision resistant is computationally infeasible to find two distinct multisets $M_1 \neq M_2$ of elements from S whose cardinalities are of polynomial size in m such that $H(M_1) = H(M_2)$, i.e., it should hold that for every probabilistic polynomial time adversary $\mathcal{A}$:*

$$\Pr\left[H(M_1) = H(M_2) | (M_1, M_2) \leftarrow \mathcal{A}^{MSH}\right] \leqslant \epsilon(\alpha, \beta, m).$$

where ϵ is negligible in α, β, and m.

2.3.2 *Encryption Schemes*

Definition 2.10 (Symmetric Encryption Scheme). *An* encryption scheme *is a tuple of probabilistic polynomial time algorithms* $(\mathsf{genkeyenc}, \mathsf{encrypt}, \mathsf{decrypt})$. *Let M denote the message space of the encryption scheme, and C denote its ciphertext space. On input of security parameter $\ell_{\mathsf{encrypt}} \in \mathbb{N}$, $\mathsf{genkeyenc}$ outputs a secret encryption key k, i.e., $k \leftarrow \mathsf{genkeyenc}(1^{\ell_{\mathsf{encrypt}}})$. On input of a message $m \in M$ and key k, $\mathsf{encrypt}$ outputs a ciphertext c in C, i.e., $c \leftarrow \mathsf{encrypt}(k; m)$. Finally, on input of ciphertext c in C and key k, $\mathsf{decrypt}$ outputs a message m such that $c = \mathsf{encrypt}(k; m)$, i.e., $m \leftarrow \mathsf{decrypt}(k; c)$. It should hold $\forall \ell \in \mathbb{N}$ that:*

$$\Pr\left[\mathsf{decrypt}(k; \mathsf{encrypt}(k; m)) = m | k \leftarrow \mathsf{genkeyenc}(1^{\ell_{\mathsf{encrypt}}})\right] = 1.$$

Definition 2.11 (Chosen Plaintext (CPA) Security). *Consider an encryption scheme as defined in Definition 2.10* $(\mathsf{genkeyenc}, \mathsf{encrypt}, \mathsf{decrypt})$ *that has a security parameter $\ell_{\mathsf{encrypt}} \in \mathbb{N}$. We define $\mathrm{Exp}_{\mathcal{A}}^{CPA-b}$ as a security experiment between a CPA-challenger denoted by C^{CPA} and adversary $\mathcal{A}$. C^{CPA} first generates an encryption key $k \leftarrow \mathsf{genkeyenc}(1^{\ell_{\mathsf{encrypt}}})$. $\mathcal{A}$ then generates two messages $m_0, m_1 \in M$ and sends them to C^{CPA}. As a next step, C^{CPA} generates the ciphertext $c_b \leftarrow \mathsf{encrypt}(k; m_b)$, where $b \in \{0,1\}$, and sends c_b back to $\mathcal{A}$. Finally, $\mathcal{A}$ has to return the bit b' which states whether C^{CPA} has encrypted m_0 or m_1. The output of this security experiment is the bit b', i.e., $\mathrm{Exp}_{\mathcal{A}}^{CPA-b} = b'$. An encryption scheme is called CPA-secure if every probabilistic polynomial time adversary $\mathcal{A}$ has negligible advantage in ℓ_{encrypt}:*

$$\mathrm{Adv}_{\mathcal{A}}^{CPA} = \left|\Pr\left[\mathrm{Exp}_{\mathcal{A}}^{CPA-0} = 1\right] - \Pr\left[\mathrm{Exp}_{\mathcal{A}}^{CPA-1} = 1\right]\right|.$$

2.3.3 *Authentication Schemes*

Definition 2.12 (Message Authentication Code)**.** *A* Message Authentication Code *(MAC) constitutes a tuple of probabilistic polynomial time algorithms* (genkeymac, mac, vermac)*. Algorithm* genkeymac *takes as input security parameter* $\ell_{\mathsf{mac}} \in \mathbb{N}$*, and outputs a secret key* k*, i.e.,* $k \leftarrow \mathsf{genkeymac}(1^{\ell_{\mathsf{mac}}})$*. The security parameter* ℓ_{mac} *determines the message space* $\mathcal{M}$ *of the MAC scheme. Algorithm* mac *takes as input message* $m \in \mathcal{M}$ *and key* k*, and outputs a MAC digest* μ *on* m*, i.e.,* $\mu \leftarrow \mathsf{mac}(k; m)$*. Finally, algorithm* vermac *takes as input MAC* μ*, message* m*, and key* k*, and outputs* 1 *if* μ *is a valid MAC digest on* m *and* 0 *otherwise, i.e.,* $\mathsf{vermac}(k; m, \mu) \in \{0, 1\}$*.*

Definition 2.13 (Signature Scheme)**.** *A* signature scheme *constitutes a tuple of probabilistic polynomial time algorithms* (genkeysign, sign, versig)*. Algorithm* genkeysign *takes security parameter* $\ell_{\mathsf{sign}} \in \mathbb{N}$ *as input, and outputs a secret signing key* sk *and a public verification key* pk*, i.e.,* $(sk, pk) \leftarrow \mathsf{genkeysign}(1^{\ell_{\mathsf{sign}}})$*. The security parameter* ℓ_{sign} *determines the message space* $\mathcal{M}$ *of the signature scheme. Algorithm* sign *takes a message* $m \in \mathcal{M}$*, and secret key* sk *as input, and outputs a signature* σ *on the message* m*, i.e.,* $\sigma \leftarrow \mathsf{sign}(sk; m)$*. Finally, algorithm* versig *takes signature* σ*, message* m*, and public key* pk *as input, and outputs* 1 *if* σ *is a valid signature on* m *and* 0 *otherwise, i.e.,* $\mathsf{versig}(pk; m, \sigma) \in \{0, 1\}$*.*

Definition 2.14 (Aggregate Signature / Multisignature)**.** *An* Aggregate (resp. Multi) signature scheme *is a tuple of probabilistic polynomial time algorithms* (genkeysign, sign, versig, aggsign, veraggsign)*. On input of security parameter* $\ell_{\mathsf{sign}} \in \mathbb{N}$*,* genkeysign *outputs a secret signing key* sk_i *and a public verification key* pk_i*, i.e.,* $(sk_i, pk_i) \leftarrow \mathsf{genkeysign}(1^{\ell_{\mathsf{sign}}})$*. On input of message* $m_a \in \mathcal{M}$ *and* sk_i*,* sign *outputs a signature* σ *on* m_a*, i.e.,* $\sigma \leftarrow \mathsf{sign}(sk_i; m_a)$*. On input* σ*,* m_a*, and* pk_i*,* versig *outputs* 1 *if* σ *is a valid signature on* m_a *and* 0 *otherwise, i.e.,* $\mathsf{versig}(pk_i; m_a; \sigma) \in \{0, 1\}$*. On input of signatures* $\sigma_1 \cdots \sigma_n$ *on distinct (resp. same) message(s),* aggsign *outputs an aggregate signature* σ*, i.e.,* $\sigma \leftarrow \mathsf{aggsign}(\sigma_1, \cdots, \sigma_n)$*. On input of* σ*,* $m_1, \cdots, m_n$ *(resp.* m_a*), and* $pk_1, \cdots, pk_n$*,* veraggsign *outputs* 1 *if* σ *is an aggregate of valid signatures* $\sigma_1, \cdots, \sigma_n$ *on* $m_1, \cdots, m_n$ *(resp.* m_a*) and* 0 *otherwise, i.e., the output of* $\mathsf{veraggsign}(pk_1, \cdots, \mathrm{pk}_n; m_1, \cdots, m_n, \sigma) \in \{0, 1\}$ *(resp.* $\mathsf{veraggsign}(pk_1, \cdots, \mathrm{pk}_n; m_a; \sigma) \in \{0, 1\}$*).*

Definition 2.15 (Selective Forgery under Chosen Message Attack (CMA))**.** *Consider a signature scheme as defined in Definition 2.12* (genkeysign, sign, versig) *that has a security parameter* $\ell_{\mathsf{sign}} \in \mathbb{N}$*. We define* $\mathrm{Exp}_{\mathcal{A}}^{\mathrm{CMA}}$ *as a security experiment between a CMA-challenger denoted by* $\mathrm{C}^{\mathrm{CMA}}$ *and adversary* $\mathcal{A}$*.* $\mathrm{C}^{\mathrm{CMA}}$ *first generates a signing key pair* $(sk, pk) \leftarrow \mathsf{genkeysign}(1^{\ell_{\mathsf{sign}}})$*, sends* pk *to* $\mathcal{A}$*, and selects a message* m^* *that it wants* $\mathcal{A}$ *to sign.* $\mathcal{A}$ *then creates a message* m *and sends it to* $\mathrm{C}^{\mathrm{CMA}}$*. As a response,* $\mathrm{C}^{\mathrm{CMA}}$ *generates the signature* $\sigma \leftarrow \mathsf{sign}(sk; m)$ *and sends it back to* $\mathcal{A}$*.* $\mathcal{A}$ *repeats this step requesting signatures on messages of its choice. Then* $\mathrm{C}^{\mathrm{CMA}}$ *sends the selected message* m^* *to* $\mathcal{A}$ *which outputs a signature* σ^**. Finally,* $\mathrm{C}^{\mathrm{CMA}}$ *returns its decision* $b = 1$ *that indicates whether* σ^* *is a valid signature over* m^**. The output of this security experiment is the decision* b *of* $\mathrm{C}^{\mathrm{CMA}}$*, i.e.,* $\mathrm{Exp}_{\mathcal{A}}^{\mathrm{CMA}} = b$*. A signature scheme is secure against selective forgery under CMA if it holds that for every probabilistic polynomial time adversary* $\mathcal{A}$*:*

$$Adv_{\mathcal{A}}^{CMA} = \Pr\left[Exp_{\mathcal{A}}^{CMA} = b\right] \leqslant \epsilon(\ell_{\mathsf{sign}}).$$

where ϵ *is negligible in* ℓ_{sign}.

Remote Attestation. Remote attestation refers to an interactive protocol executed between two parties: (1) the verifier V, and (2) the prover P. The goal of RA is to allow V to remotely check and verify the software integrity of P, and hereby detect compromise (i.e., malicious code modification) on P. RA is initiated by V sending an attestation request *req* including a random challenge N to P. N is required to thwart replay attacks on RA. In response to an attestation request, a trusted component on P, denoted by *attestor* measures P's software state by generating a cryptographic hash of its binaries. Attestor is typically established using hardware security architectures [52, 57]. The measurement is authenticated by attestor and reported back to the V as an attestation response *resp*. Based on the set of benign software states, V can then determine whether P is trustworthy.

3

RELATED WORK

The pervasiveness of embedded devices and their increasing sensing and actuation capabilities has made them attractive targets for attacks. Malware infestation represents one sevre class of attacks that threatens both privacy and safety of the users of such devices, e.g., stuxnet [148]. As a consequence, security solutions which defend against malware infestation attacks have been increasingly demanded. One prominent solution for detecting such attacks is secure software attestation. Attestation allows a trusted verifier to check the integrity of the software on a remote prover. It has been a popular research topic in established security conferences over the course of the last two decades. Attestation is usually established as an interactive protocol, where a remote entity (the verifier) sends a challenge to an untrusted device (the prover). As a response, the prover generates a measurement of its software state and sends it back to the verifier.

In this chapter, we provide background information and a systematic study of existing work on attestation. In particular, we first introduce different attacks that are relevant for attestation, and describe requirements imposed by existing attestation solutions as well as the security properties they provide. We then classify these solutions based on their requirements, security properties, and the attacks they detect or are vulnerable to. In general these solutions can be clustered into three main classes: (1) software-based attestation that requires no secure hardware whatsoever, (2) hardware-based attestation that requires complex and expensive security hardware, and (3) hybrid attestation that requires a small set of security features in hardware. Additionally, a fourth class of attestation schemes has recently emerged which aims at detecting runtime attacks through attesting the control flow of an executing software rather than its binaries. This class is denoted by (iv) Control-Flow Attestation (CFA). Security of CFA is based on the hardware features required by hybrid attestation.

3.1 ATTESTATION LANDSCAPE

The goal, scope, and security properties of remote attestation has evolved throughout the years allowing better security gaurantees (e.g., TPM [144]), requiring less complex hardware (g., SMART [52]), or detecting more sophisticated attacks (e.g., Control-Flow Attestation – CFA [9]). Similarly, various attacks that target existing attestation schemes have been devised. In this section we introduce the four identified attestation classes and present all attacks that are relevant to attestation, i.e, attacks that either target or are detected by existing attestation schemes.

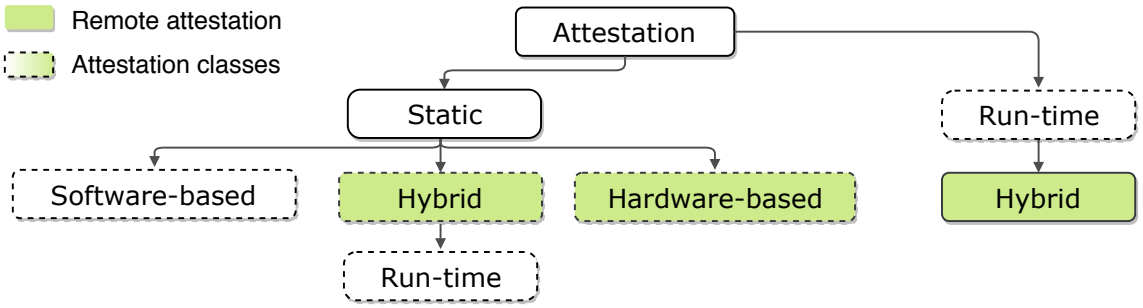


Figure 3.1: Taxonomy for attestation

3.1.1 *Taxonomy of Attestation*

We classify existing attestation schemes based of multiple factors. In particular, based on security guarantees provided by these schemes and on their requirements in terms of hardware and cryptographic secrets, these schemes can be classified into software-based attestation, hardware-based attestation, and hybrid attestation. Furthermore, these schemes can be classified based on the nature of the attacks that they detect into static and run-time attestation. While static attestation measures the software binaries at load time, runtime attestation measures the software execution at runtime. It is important to note that except for software-based attestation, attestation solutions can typically be executed remotely and are hence referred to as remote attestation schemes. Figure 3.1 shows the taxonomy of attestation: The figure shows the four main classes that existing attestation solutions are clustered into as well as the relation between different classes. For example, runtime attestation schemes require basic security features in hardware and are considered as a subclass of hybrid attestation. Similarly, software-based attestation, hardware-based attestation, and hybrid attestation are considered static attestation solutions. In the following we briefly introduce each of these classes:

Software-based Attestation. Software-based attestation [81, 126, 125, 124, 122, 62, 89] refers to attestation solutions that require no hardware security or cryptographic secrets. The security of these solutions is rather based on the limited computational power of the prover and on strict estimation of the time required to execute the attestation protocol. Not relying on the existence of security hardware has made software-based attestation an interesting approach for verifying the integrity of low-end embedded and legacy computing devices. Unfortunately, the security of software-based attestation is based on strong assumption that are hard to achieve in most realisitc settings [15], such as adversarial silence during attestation and optimality of attestation protocol and its implementation. As a consequence, existing software-based solutions were targeted by many successful attacks [130, 32, 151] and their applicability was limited to attesting peripherals [89].

Hardware-based Attestation. Hardware-based attestation [108, 52, 86, 119, 85] refers to attestation solutions that require complex hardware such as a secure co-processor (e.g.Trusted Platform Modules – TPM [144]) or extensive hardware modifications. These

hardware requirements provide a secure trust anchor that ensures the integrity of code used for attestation and the secrecy of the cryptographic keys used for authentication. They hereby provide strong guarantees on the security of attestation. However, such hardware requirements are too complex and expensive for low-end embedded devices.

Hybrid Attestation. Hybrid attestation [52, 57, 84, 28] refers to attestation solutions that provide strong security guarantees (comparable to those of hardware-based attestation) while minimizing the hardware requirements. This is achieved by leveraging a software/hardware co-design, which allows required hardware features to be as simple as a Read Only Memory (ROM) and a simple Memory Protection Unit (MPU). Although minimal, the hardware features required by hybrid attestation solutions render them inapplicable to legacy computing devices.

Runtime Attestation. Runtime attestation [46, 20, 83, 65, 155, 9, 49] refers to the class of attestation solutions that capture the behavior of an executing program at runtime rather than measuring its binaries at load time. In particular, existing runtime attestation solutions focus on measuring certain aspects of a program's execution, e.g., the base pointer integrity of called functions, the launch order of program's modules, and various other properties that may present an indication of runtime attacks. Recent effort on runtime attestation devised schemes that are capable of reporting the exact control-flow path of an executing program to a remote verifier. These schemes are also referred to as Control-Flow Attestation (CFA) schemes.

3.1.2 *Taxonomy of Attacks*

As the goal and the security guarantees of attestation evolved: several attacks have been successfully performed on existing attestation schemes, several new attacks became detectable by attestation, and various attacks were added to the threat model of attestation. Figure 3.2 shows all these attacks and their relation to the four aforementioned attestation classes. In the following we describe each of the individual attacks. We denote an adversary by $\mathcal{A}$.

Communication Attacks. In a communication attack, $\mathcal{A}$ possess full control over all communication channels, i.e., $\mathcal{A}$ is capable of eavesdropping on, dropping, modifying, and delaying all exchanged packets as well as injecting its own packets on the communication channel. Communication attacks cannot be detected by attestation. However, they are within the threat model of most existing attestation schemes. These attacks are usually thwarted by means of authentication.

Malware Attacks. In a malware attack, $\mathcal{A}$ compromises a device by maliciously modifying its software state. $\mathcal{A}$ then injects its own code (i.e., malware) on that device. Malware injection allows $\mathcal{A}$ to execute an arbitrarily malicious functionality on the infected device, for example, $\mathcal{A}$ could read sensitive data that is not protected by hardware, report wrong information, or perform malicious actions. The initial goal of attestation has been the detection of such malware infestation attacks.

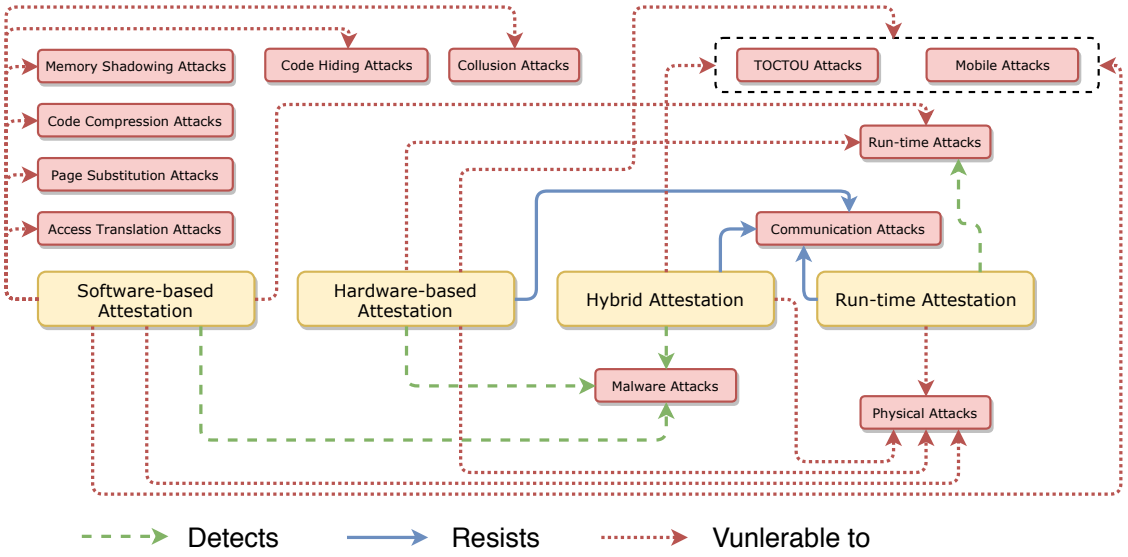


Figure 3.2: Taxonomy for attacks

Physical Attacks. In a physical attack, $\mathcal{A}$ exploits its physical presence next to a device in order to extract its secrets or modify its software or hardware components, e.g., overclocking the processor or replacing a memory module. We distinguish between three classes of physical attacks: (1) invasive attacks [133] that require $\mathcal{A}$ to access a device's internal components using sophisticated lab equipment, (2) semi-invasive attacks [134] that require $\mathcal{A}$ to at least perform decapsulation of the device using less complicated tools, (3) non-invasive attacks [160] that allow $\mathcal{A}$ to extract device's secrets during normal operation using low-cost engineering tools. Physical attacks are considered out of scope of most existing attestation schemes that typically assume a software-only adversary. Consequently, an adversary that is capable of performing physical attacks can undermine the security of all these schemes.

Runtime Attacks. In a runtime attack, $\mathcal{A}$ exploits a software vulnerability, e.g., a buffer overflow, to perform a malicious behaviour during program execution [140]. For example, a buffer overflow vulnerability in a program can be exploited by $\mathcal{A}$ to overwrite memory cells adjacent to the buffer, which contain control-flow information, e.g., a functions return address. This allows $\mathcal{A}$ to hijack the control- low of the program and jump to an arbitrary address of choice. The most prominent example is Return-Oriented Programming (ROP) attack [127], through which $\mathcal{A}$ stitches together small snippets of machine code, already residing on device's memory, to execute an arbitrarily malicious functionality. We identify two classes of runtime attacks that are relevant to existing work on attestation: (1) non-control-data attacks that modify non-control data indirectly affecting a program's control flow, e.g., branch variables, and (2) control-data attacks that overwrite functions pointers and return addresses.

Proxy and Collusion Attacks. In a proxy/collusion attack, $\mathcal{A}$ evades detection by software-based attestation by delegating or speeding up the execution of the attestation protocol. In particular, in order to hide compromise of a device, the attestation is either delegated

to an identical benign device, simulated by a more powerful device, or executed in parallel by multiple devices. All existing software-based attestation schemes are vulnerable to proxy attacks, since the verifier in such schemes is incapable of authenticating the prover.

Code Hiding Attacks. In a code hiding attack, $\mathcal{A}$ moves malicious code between program and data memory in order to evade detection by software-based attestation [32]. Consequently, attestation is performed on benign code residing in program memory, while the malicious code resides on the device's data memory. After execution of attestation, $\mathcal{A}$ moves malicious code back to program memory for execution. Note that, performing code hiding attack could be based on ROP that allows stitching together code snippets required for hiding and restoring the malicious code.

Address Translation Attacks. In an address translation attack, $\mathcal{A}$ evades detection by software-based attestation through exploiting the address translation mechanism of modern CPUs [151]. In particular, since modern processors enforce the distinction between code and data, $\mathcal{A}$ is capable of allowing a device to attest to a different code than the one it executes. In order to execute this attack, $\mathcal{A}$ is required to modify the device's operating system which is responsible for virtual address translation. This allows $\mathcal{A}$ to maliciously modify and execute a target program, while redirecting data accesses to an unmodified copy of this program.

Page Substitution Attacks. In a page substitution attack, $\mathcal{A}$ evades detection by software-based attestation through adding malicious attestation code at the same memory page containing the measurement code [130]. This malicious attestation code allows $\mathcal{A}$ to convince the verifier of the trustworthiness of a prover despite its existence in the memory. This leads to a successful attestation of the prover, while at the same time allowing for arbitrary subsequent modification to the its memory. In the case where the page holding the measurement code is full, $\mathcal{A}$ may exploit code compression (see below) in order to perform a page substitution attack.

Code Compression Attacks. In a code compression attack $\mathcal{A}$ evades detection by software-based attestation by compressing the code on the prover [32]. In particular, $\mathcal{A}$ compresses the benign code in the memory to generate enough space in the program memory for storing malicious code. At attestation time benign code is decompressed and measured on-the-fly. Code compression attacks target software-based attestation schemes that are based on filling the memory of a device with randomness in order to leave no space for malicious code (see Section 3.4.2).

Memory Shadowing Attacks. In a memory shadowing attack $\mathcal{A}$ evades detection by software-based attestation by exploiting empty memory on the prover. In particular, $\mathcal{A}$ compromises a device by injecting malicious code in an empty memory position. At attestation time, $\mathcal{A}$ simply evades detection by redirecting memory reads that target malicious code to another empty shadow memory, i.e., a memory that has the same address with only one single bit flipped. Flipping only one single bit leads to a negligible increase in the execution time of the attestation protocol. This negligible increase is not detected by existing software-based attestation solutions that are based on measuring the attestation time (see Section 3.4.1).

TOCTOU Attacks. In a Time of Check Time of Use (TOCTOU) attack, $\mathcal{A}$ exploits the time difference between that attestation of a program and its execution in order to perform a malicious functionality on a device without being detected by the attestation protocol. In other words, $\mathcal{A}$ introduces and executes malicious code before or after attestation, while restoring the code to its benign state during attestation. TOCTOU attacks are very hard to protect against and are not considered by most existing attestation protocol. Such attacks can also be based on physical [155] or runtime attacks [32].

3.2 REQUIREMENTS FOR ATTESTATION

Existing attestation schemes have different requirements that separate them from each other and determine their applicability to different scenarios. We distinguish between three types of requirements of an attestation solution: software, hardware, and cryptographic requirements.

3.2.1 *Hardware Requirements*

Hardware requirements range from requiring no hardware, which applies to software-based attestation solutions, to requiring a cryptographic co-processor, which is the case in most hardware-based attestation solutions. Hardware requirements of existing attestation solutions are presented in the following:

No Hardware Requirements. This entails not requiring any specific hardware for enabling attestation, i.e., no new hardware modules are required, and no modification to existing hardware is needed. A direct consequence of having no hardware requirements is applicability to low-end and legacy computing devices.

Complex Hardware Requirements. This entails requiring additional hardware components or applying complex and expensive modifications to existing hardware. Examples of complex hardware requirements include cryptographic co-processors such as Trusted Platform Modules (TPM), and hardware extensive modifications that induce a very large fingerprint, i.e., area overhead, or changes to the core of the CPU, e.g., extending the instruction set. A direct consequence of having complex and expensive hardware requirements is inapplicability to low-end and legacy computing devices.

Simple Hardware Requirements. This entails requiring simple hardware extensions that are neither complex nor expensive. Examples of simple hardware extensions include adding simple hardware modules, or performing small hardware modifications that have a small fingerprint. Such modification usually aim at providing simple security features, e.g., a Read Only Memory (ROM) that ensures integrity of the code and data its stores, or a simple Memory Protection Unit (MPU) that restricts access to a certain memory area, thus, providing integrity and/or confidentiality of data. Requiring only simple hardware allows attestation schemes to be applied to low-end but not legacy computing devices.

Other Hardware Requirements. This refers to other hardware-related requirements that are imposed on the prover or the verifier of an attestation schemes. Examples of such requirements include prior knowledge of prover's hardware configurations by the verifier, e.g., knowledge of memory size or clock rate of the prover's CPU.

3.2.2 *Software Requirements*

Software requirements refer to requirements that are either imposed on the attestation code, i.e., software used for executing the attestation protocol; or the attested software, i.e., the software whose integrity is to be verified. The former is mainly relevant for software-based attestation, while the latter is relevant for runtime attestation. Software requirements of existing attestation solutions are presented in the following:

Access to Source Code of Attested Software. This entails knowing the source of the software on the prover that needs to be attested. Access to source code is usually relevant for runtime attestation, where the source code is required to extract properties regarding the its execution behavior, e.g., Control-Flow Graph (CFG). Requiring access to source code hinder applicability to legacy or closed-source code which is usually only available in binary format.

Instrumentation of Attested Software. This entails applying modifications to the software on the prover that needs to be attested. Examples of instrumentation include adding specific instructions to the code which are crucial for the execution of the attestation protocol. Instrumentation can be either performed at compile time provided access to source code, or can be based on binary instrumentation otherwise. This requirement is usually relevant for runtime attestation which expects the attestation code to be triggered continuously throughout the executions of the attested software.

Optimality of Attestation Code. This entails requiring ensuring that the code responsible for executing the attestation protocol on the prover is optimal in terms of time. Optimality requirement should apply to both the attestation algorithm and its implementation. In other words, finding a different algorithm or implementation that executes the attestation protocol in less time than the original attestation code should not be possible. This requirements is relevant for software-based attestation, whose security is based on the exact estimation of the time required by the prover to execute the attestation protocol.

3.2.3 *Cryptographic Requirements*

Cryptographic requirements of attestation range from having no cryptographic requirements, which applies to software-based attestation solutions; to requiring a key management scheme and cryptographic secrets; which is the case for hardware-based attestation solutions. Cryptographic requirements of existing attestation solutions are presented in the following:

No Cryptographic Requirements. This entails not requiring any cryptographic secrets or key managements protocols. Schemes that have no cryptographic requirements usually rely on an out of band channel in order to allow authentication the prover.

Cryptographic Secrets. This entails requiring the prover to store one or more cryptographic keys that are used for authentication. Cryptographic secrets can be either a public key pair, or a symmetric key that is shared with a trusted verifier.

Key Management Protocols. This entails requiring means for sharing symmetric keys or prove possession and security of private keys. Key management can be handled using public key certificates based on existing public key infrastructure (PKI), which is the case for TPMs.

3.3 PROPERTIES OF ATTESTATION

Existing attestation schemes have different performance- and security-related properties. These properties distinguish the schemes from one another. They play an important role in determining the applicability of each scheme in different scenarios.

3.3.1 *Performance Properties*

Performance properties of attestation are mainly concerned with the overhead imposed on each of the prover and the verifier of an attestation scheme. This also concerns the scalability of the scheme and its applicability to large networks. Performance properties of existing attestation solutions are presented in the following:

Efficiency on the Prover's Side. This entails requiring the attestation solution to impose low overhead on the prover in terms of energy consumption and runtime. Importance of runtime efficiency on prover is significant since existing attestation schemes usually require uninterrupted execution of the attestation protocol. This results in taking the prover away from performing its original task. In software-based attestation the high runtime overhead on the prover is basically due to the very large number of memory access operations. These operation are required to ensure that every memory cell is accessed (with a very high probability) during random traversal. On the other hand, runtime attestation induce a high runtime overhead on the prover due to continuous generation of hash values at runtime. Finally, energy consumption on the prover is also of particular importance due to the limited sources of energy on low-end embedded devices. These devices are usually battery-powered.

Efficiency on the Verifier's Side. This entails requiring the attestation solution to impose low overhead on the verifier in terms of energy consumption and runtime. The verifier in an attestation protocol can either be a powerful device, e.g., a server, or a less powerful entity, e.g., a low-end embedded device with limited resource. In the latter case, overhead on the verifier is particularly important. In software-based attestation the high runtime overhead on the verifier is mainly due to the recalculation of the attestation response. This requires the verifier to simulate the execution of the attestation code

on the prover. On the other hand, runtime attestation induce a high runtime overhead on the verifier, which is required to search a very large database of benign attestation responses.

Scalability. This entails applicability to a very large number of provers. In particular, we consider an attestation solution to be scalable if the increase in its runtime is (at most) logarithmic in the number of devices that are attested. Existing attestation solutions have a runtime overhead that increases linearly with the number of provers when naively applied in a multi-device setting.

3.3.2 *Security Properties*

Security properties of attestation are mainly concerned with the inclusion of certain attack vectors in the threat model and providing resiliency against these attacks. Security properties of existing attestation solutions are presented in the following:

Time Consistency. Time consistency refers to ensuring that the generated measurement over a certain software (or piece of memory) indeed reflects the state of all parts of the software (or content at different memory addresses) at the exact same time period. Such property prevents an adversary from evading detection by moving malicious code within the prover's memory during the execution of the attestation code. The most natural technique for imposing such a property is locking the prover's memory throughout the attestation process.

Hardware Independence. Hardware independence entails that the security of the attestation solution is independent of prover's hardware properties. Such hardware properties may include the size of the memory to be attested, or the clock speed of the prover's processor. Attestation solutions that are not hardware independent, e.g., software-based attestation are vulnerable to physical attacks that aim at extending the prover's memory or overclocking its processor.

End-to-End Security. End-to-end security entails resistance against man-in-the-middle attacks and hereby applicability in remote settings. Such property is established through the authentication of the software measurements taken at the prover. Authentication is usually based on cryptographic secrets that protected by the prover's hardware.

Resiliency Against TOCTOU. Resiliency against TOCTOU entails requiring that the software executed on the prover is identical to the attested one. Such property prevents an adversary from compromising a device's software state in the time gap between attestation and execution. Most existing attestation solutions are not resilient against TOCTOU attacks. These attacks can be either launched remotely, i.e., by software means, or via hardware means.

Detection of Runtime Attacks. This property entails the detection of attacks that do not change the binaries of the software on the prover, but rather its runtime behavior during exection, e.g., Return-Oriented Programming (ROP) attacks [127]. While runtime attestation aim at detection of such runtime attacks, static attestation solutions are in fact vulnerable to these attacks, that they usually consider to be out of scope.

3.4 SOFTWARE-BASED ATTESTATION

The goal of software-based attestation is to allow software integrity verification without requiring hardware security features. This is mainly driven by the need to reduce the deployment and upgrade costs as well as allowing applicability to legacy and low-end computing devices. The core of most existing software-based attestation schemes is a so-called checksumming or verification function, which generates the measurement of prover's software state. The checksumming function is carefully designed to utilize all memory resources and computational power of the prover in such a way that a measurement can only be generated in time if it reflects the correct software state of the prover. As a result, the verifier only accepts a correct measurement if received within a certain time limit. Software-based attestation of this kind is also referred to as time-based attestation.

Security of time-based attestation relies on strong assumptions that are hard to achieve in most realistic scenarios. In particular, it requires adversarial silence during the execution of the attestation protocol, it assumes optimality of the checksumming function and its implementation, it requires a strict estimation of the round trip time between the verifier and the prover, and it requires that the verifier has prior knowledge of prover's hardware configurations and memory contents. The checksumming function usually operates in a loop, where one random memory address is accessed at each iteration and incorporated into the measurement. This random memory traversal can be based on the random challenge sent by the verifier.

Some of the existing software-based attestation schemes are not based on the execution time of the attestation code, but rather on exploiting memory constraints of the prover. In particular, such schemes protects the prover against malware infestation by filling its unused memory with randomness. This leaves no free space for the adversary to inject malicious code.

3.4.1 *Time-Based Attestation*

In the following we present existing time-based attestation schemes for both low-end and advanced computing devices.

Genuinity. Genuinity [81] is a time-based attestation solution that allows verifying the integrity of the operating system and the hardware of a legacy computing device. The core of Genuinity is a checksumming function that incorporates the side effects of its execution in calculating the measurement of a device's software state. As a result, a device that has different operating system and/or hardware would consume more time in order to generate a correct measurement. Examples of the side effects incorporated in the measurement in Genuinity include: performance counters, cache tags, and cache miss counts of the Translation Lookaside Buffer (TLB). Genuinity assumes that the verifier has prior knowledge of the prover's hardware configuration. It also assumes a platform with a single processor that is not modified by the adversary.

Pioneer. Pioneer [125] aims at providing verifiable code execution for legacy computing devices, i.e., it ensures that a piece of untampered software was successfully executed. Pioneer is based on a checksumming function that is capable of verifying its own integrity as well as the integrity of other executables. In particular, this function first generates a measurement over its own code and sends it to the verifier. After verifying the integrity of the checksumming function, it can be leveraged to establish a dynamic root of trust on the prover through measuring every executable before invoking it uninterrupted. In order to ensure security, Pioneer assumes that interrupts are disabled during the execution of the checksumming function. PioneerNG [121] extends Pioneer by relaxing its assumptions. This is done by including more system state components in the generation of the software measurement. Finally, Checkmate [86] provides a dynamic root of trust that is resilient to TOCTOU attacks based on PioneerNG's checksumming function. In particular, Checkmate combines Address Space Layout Randomization (ASLR) with PionerNG's checksumming function. This function is indeed modified to tolerate ASLR.

SWATT. SWATT [126] is another time-based attestation scheme for embedded devices. In SWATT random memory addresses traversed by the checksumming function are generated using Pseudorandom Number Generator (PRNG) seeded by the challenge received from the verifier. In fact, SWATT identifies the properties that should be satisfied by this function in order for software-based attestation to be secure. The checksumming function should be non-parallelizable, and resistant to replay and pre-computation attacks. Moreover, it should have a small code size, perform memory accesses in a pseudorandom manner, and generate (with high probability) an invalid measurement if a single memory location was modified. Satisfying these properties would lead to a noticeable increase in the function's execution time if the adversary tries to hide modifications to the prover's memory. SBAP [88] presents an optimization of SWATT that allows its application to low-end microcontrollers such as peripherals. This is done by replacing SWATT's PRNG with a more efficient one. Further, SBAP proposes thwarting Return-Oriented Programming (ROP) attacks on software-based attestation by measuring both data and program memory. It also fills the unused memory of the prover with randomness in order to leave no free space for the adversary to exploit. Finally, VIPER [89] also aims at verifying integrity of peripherals' firmware. It improves the security of previously proposed software-based attestation schemes by mitigating proxy attacks. In particular, VIPER thwarts onboard proxies by attesting all peripherals in decreasing order of their computing power. It also prevents remote proxy attacks by increasing the size of the verifier's challenge or performing multiple iteration of the attestation protocol.

ICE. ICE [124] aims at providing untampered code execution on low-end embedded devices that have a simple processor architecture. In order to thwart TOCTOU and memory shadowing attacks, the checksumming function of ICE incorporates the data pointer, program counter, processor's interrupt vector table, and interrupt disable bit in the generation of the software measurement. SCUBA [124] exploits ICE to enable secure software update for sensor nodes in Wireless Sensor Networks (WSN). FIRE [123] presents a suite of protocols that are capable of detecting node compromise in WSN, while enabling revocation or recovery of malicious nodes. In particular, ICE's checksum-

ming function is used to verify its own integrity as well as the integrity of one of the FIRE protocols. The verified protocol is then executed uninterrupted. In contrast to ICE, the checksumming function in FIRE performs multiple linear memory traversals instead of a random one. Moreover, in order to enable software integrity verification of remote devices, FIRE proposes using the expanding ring method, where every node recursively attests its neighbors until the targeted node is reached. Finally, SAKE [122] uses ICE to allow secure key establishment in WSN. In particular, by ensuring untampered execution of SAKE's code on sensor nodes involved in sharing a key, ICE guarantees that the shared key is not revealed to any malicious code on any of these nodes. SAKE uses embedded silicon IDs to achieve authenticity of the involved nodes. The correct reading of these IDs is also guaranteed by ICE.

TEAS. The following time-based attestation schemes require the verifier to construct and send the prover a new checksumming function or a Time Executable Agent System (TEAS) for each execution of the attestation protocol. TEAS [60] hides the critical functionality of the checksumming function within randomly generated code. Further, it thwarts online/offline analysis of checksumming functions by imposing strict time limits on its execution, leveraging code obfuscation, and providing a very large library of different functions. In particular online analysis is prevented by permuting the prover's memory content before each attestation. On the other hand, offline analysis on the checksumming function is prevented by increasing its complexity, e.g., adding jumps into a loop body. This complicates the construction of a Control-Flow Graph (CFG) by making it super-linear in space and time. Shaneck et al. also proposes a software-based attestation solution, where the verifier constructs a randomized checksumming function for each protocol execution [129]. This solution exploits code obfuscation techniques such as junk instruction, opaque predicates, and self-modification to thwart attacks that are based on static and dynamic analysis of the verification function. The main motivation behind this is to enable software-based attestation to be applied remotely by making its security less dependent on the execution time of the checksumming function.

3.4.2 *Untimed Software-Based Attestation*

This refers to attestation solutions that are not dependent on the execution time of the verification function. They are mostly based on leaving no free space for storing malicious code on the prover by filling its memory with randomness. Security of untimed software-based attestation schemes is based on memory constraints of the prover and on the incompressibility of randomness.

Filling Memory. Choi et al. propose an untimed software-based attestation solution for Wireless Sensor Networks (WSN) that is based on clearing all the free space on a prover's memory before the execution of attestation [38]. In particular, prior to measuring the software on the prover, its unused memory space is filled with pseudorandomness, which is generated based on a random seed included in the challenge received from the verifier. Cao et al. present a distributed untimed software-based attestation scheme [152]. In this scheme the neighbors of the prover collaborate to attest its software. In particular, before

deployment the unused space in a sensor's memory is filled with pseudorandomness. To attest a sensor node, the neighbors collaborate to reconstruct the content of its unused memory and verify its software measurement. The software measurement in this scheme is generated based on random memory traversal.

Reflection. Reflection [136] denotes the process through which a software reads and measures its own code using cryptographic hashes. It is used to attest a device's software state. Reflection assumes that the memory of a device has no space with low entropy, i.e., it assumes that unused memory space is filled with randomness. In order to measure the software state of a device, its memory is split into two overlapping regions, which are then measured and reported to the verifier. The boundaries of these regions are determined within the challenge sent by the verifier and are different for each execution. To account for compressibility of code and low entropy of memory content the prover also incorporates processor state in the generation of the measurement. Although the security of reflection is not based on the time required to generate a software measurement, the authors suggest using timing for detection of relay attacks.

PIV. PIV [107] is a software-based attestation scheme for verifying the software integrity of sensor nodes in WSN. Similar to TEAS, in PIV the verifier creates a new checksumming function for each execution of the attestation protocol. However, PIV does not impose any constraints on the time required by the prover to measure its software. PIV is based on so-called Randomized Hash Functions (RHF), which are created using multivariate quadratic polynomials and aim to replacing keyed hash functions. In particular, for each attestation the verifier creates a different randomized hash function and sends it to the prover as the attestation challenge.

3.5 HARDWARE-BASED ATTESTATION

While software-based attestation is based on strong assumptions and cannot be applied remotely, hardware-based attestation aims at providing secure remote attestation with strong security guarantees. Hardware-based attestation refers to attestation solutions that rely on hardware modifications or secure co-processors that provide authentication and integrity verification. Note that authentication and integrity verification are considered the enablers for secure remote attestation. Security hardware usually provide trusted execution of the attestation code, and secure storage for sensitive data such as cryptographic secrets. The underlying security architecture of hardware-based attestation is too complex and expensive. Therefore, these schemes are suitable for high-end computing devices, e.g., laptops, servers, and smartphones.

3.5.1 *TPM and TPM-based Attestation*

Secure Boot. Secure boot [14] guarantees that a computer comes up to a secure and reliable software state directly after boot. It is based on a chain of trust built starting at the Basic Input/Output System (BIOS) and ending at the operating system. In particular, when a computer is powered up the integrity of each software is verified prior to its

execution by the software that passes control to it. Integrity verification is based on comparing a cryptographic hash of the binaries of the loaded software to a digitally signed reference value. Integrity of the software that is first to load is ensured through hardware, e.g., by storing it in Read-Only Memory (ROM). If the integrity verification failed, e.g., due to a modified software component, the systems boots a recovery kernel which contacts a remote host and requests the original software component. Integrity of recovery kernel is also ensured through ROM.

TPM. Trusted Platform Module[1] (TPM) represents a specification of a cryptographic co-processor that provides integrity protection and handles cryptographic operations. As specified by the Trusted Computing Group (TCG) [144], TPM contains a group of Platform Configuration Registers (PCR) that are only resettable upon reboot. These PCRs are extended with the measurements (i.e., cryptographic hash) of every software executed on a device. Similar to secure boot, a chain of trust is built by TPM by extending the value of a certain PCR with the hash of every software (starting with BIOS) before transferring control to it. This is usually referred to as static root of trust or trusted boot. However, unlike secure boot, TPM records the execution of modified code but does not prohibit it. In order to provide remote attestation, TPM allows signing a PCR value along with a received challenge (i.e., random nonce) based on the TPM's securely stored private key. TPM v1.2 [143] allows establishing a dynamic root of trust. It provides the specifications for a dynamic PCR that can be reset using a dedicated instruction. Intel and AMD provide a late lunch functionality through the instructions SKINIT and SENTER respectively, which reset the value of PCR 17, and store the measurement of the executed software into that PCR. Similarly, the value stored in PCR 17 can be signed using TPM's secret key providing remote attestation.

TCG-based Attestation. TCG-based attestation [117] aims at extending TPM's chain of trust to the application layer. In particular, it presents modifications to the kernel that allow it to measure itself (i.e., the loadable kernel modules) as well as executed applications and extend this measurement to a certain PCR. To allow verification of the PCR's content, the kernel also keeps a log of all applications and modules that were measured and extended to the PCR. During attestation, the kernel sends the content of the PCR signed by TPM along with the log file to the remote verifier.

OSLO. Due to multiple vulnerabilites identified in TPM's implementation of static root of trust, OSLO [79] exploits AMD's SKINT and TPM to provide a dynamic root of trust. Identified vulnerabilities include: freely batchable BIOS, buggy bootloader that allows executing code without extending it to PCR, and hardware resettable PCRs in TPM v1.1. In OSLO, SKINT is used to reset PCR 17 and extend the measurement of a secure loader to that PCR. When executed, the loader provides a dynamic root of trust on the device by measuring and keeping track of all executed software. However, OSLO does not allow reporting recorded measurements to a remote verifier.

TPM for Timestamping. Schellekens et al. propose using TPM's timestamping functionality to improve the security guarantees of time-based attestation and enable its applicability in remote settings. In particular, TPM's timestamping is used for determining

1 https://trustedcomputinggroup.org/tpm-main-specification

the exact execution time of Poineer's checksumming function [125], thus eliminating the non-deterministic network delay. Unlike TPM's attestation, combing TPM with software-based attestation does not require trusting the operating system. However, it provides the same weak security guarantees of software-based attestation while requiring complex hardware.

Terra. In Terra [63], a general purpose platform is partitioned into multiple (general and special purpose) virtual machines (VM), which are isolated based on a Trusted Virtual Machine Monitor (TVMM). This isolation allows multiple applications with different security requirements to execute concurrently. Integrity and confidentiality of VM's data and code is provided based on cryptography and secure hardware. Terra is based on certificate chains that bind public keys to hashes of programs that possess their corresponding private keys. A certificate chain starts with TPM's hardware and ends at the involved VM. It allows a VM to securely communicate and attest its software to a remote entity. Security of remote attestation in Terra relies on TPM's secure storage and attestation.

Nizza. Nizza [66] is a security architecture that provides isolation of software components on a device based on TPM and L4 microkernel. Nizza executes software components as threads and exploits address space separation to enable their isolation. Further, it relies on Inter-Process Communication (IPC) to provide communication between isolated components. Authenticity of executing software components is assured based on TPM's trusted boot, which also allows remote attestation of these components. Singaravelu et al. exploits Nizza's isolation to allow reducing the Trusted Computing Base (TCB) of security sensitive operations. To achieve this, the authors propose extracting the security sensitive parts of a program and executing them isolated based on Nizza. The remaining of the program can be executed within the legacy operating system.

Flicker and TrustVisor. Flicker [97] presents another isolation architecture based on TPM. Flicker aims at executing small chunks of security sensitive code denoted by Piece of Application Logic (PAL) in isolation of the rest of the software components on the device. A Secure Loader Block (SLB) formed of 250 lines of code is the only software TCB of Flicker. The main task of SLB is to prepare a protected execution environment before executing a PAL and deleting all traces after a PAL's execution terminates. Flicker is based on late launch provided by Intel's Trusted Execution Technology (TXT) through the SKINIT instruction, and by AMD's Secure Virtual Machine (SVM) through the SENTER instruction. It allows attesting isolated code as well as providing a proof of secure execution of that code. Attestation, secure multitasking, and establishment of secure channels are enabled by Flicker based on TPM's attestation and secure storage. Moreover, minimal modifications to Intel's TXT and AMD's SVM were identified in [96] to enhance Flicker's performance. In particular, these modification enable hardware context switching and memory isolation, through a single instruction, multiple additional PCRs, and an access control table. Finally, in order to further improve the efficiency of Flicker, TrustVisor [95] implements some of TPM's functionality in software and adds it to Flicker's TCB. TrustVisor exploits hardware visualization to enable isolation while supporting attestation based on a chain of trust that starts at TPM, i.e., TPM measures

TrustVisor's software, which in turn measures the software of concerned PALs. TrustVisor also incorporates a mechanism for registering PALs which required custom compilation in Flicker.

SICE. SICE [17] presents an isolation architecture for x86 multicore hardware platforms. The TCB of SICE is formed of the hardware, the BIOS, and a small piece of software denoted by the System Management Mode (SMM). SMM resides on the System Management Random Access Memory (SMRAM) of the x86 architecture, and is responsible for providing secure attestation. One motivation behind the design of SICE is that SMRAM can only be triggered by system management interrupts and can be managed by applying modifications to only two registers. Further, time sharing between isolated software can be done by simply resizing SMRAM. This leads to fast context switching. Based on TPM's attestation, SICE is capable of attesting the integrity of its own code or the code of isolated software, and establishing secure communication to a remote entity.

Logic for TPM. Datta et al. present a logic that allows proving secuirty of protocols as a high level of abstraction regardless of the actions taken by the adversary [45]. The authors were capable of modeling several features through their model, e.g., machine reset, dynamic code loading and execution, access control, shared memory, and cryptographic operations. Finally, the security of two remote attestation protocols based on TPM was proven based on this logic. In particular, constructs of this logic were used to characterize primitives of trusted computing, which allowed extracting assumptions required for securing these protocols.

3.5.2 *Others*

This refers to attestation solutions that require complex hardware but are not based on TCG's TPM.

Copilot. In order to detect malicious modifications to kernels on commodity device, e.g., rootkit installations, Copilot [111] allows periodic monitoring of the software integrity of these kernels. Copilot resides on a PCI card providing a secure co-processor. It exploits specific PCI bus features of IBM and Linux virtual memory to detect kernel modifications. In particular, Copilot periodically calculates a hash of the kernel code and compares it to a reference value. Detected modifications are then reported to a remote entity for manual inspection through an out of band communication channel. Copilot accesses the kernel code via Direct Memory Access (DMA) without the processor's intervention.

Proxos. Proxos [142] aims at providing isolation of security sensitive code from the operating system based on a Xen Virtual Machine Monitor (VMM). Unlike Flicker, Proxos does not require developers to specify the security sensitive parts of their applications. It achieves isolation by partitioning application's system calls and forwarding them to either a trusted or an untrusted operating system. This partitioning is determined by routing rules specified by the developer. The VMM guarantees process isolation. However, calls routed to the untrusted operating system are routed through inter-VM remote procedure call.

TEM. TEM [43] is a scheme that allows execution of encrypted software within a trusted environment. The main goal of TEM is to provide both integrity and confidentiality of software executing on a computing device. In order to provide confidentiality, data provided to TEM are encrypted based on TEM's public key. This data is also stored at randomized memory addresses. The integrity of data is provided based on Merkle Hash Trees (MHT). Finally, secure execution of software is enabled by executing one software at a time. The intermediate results of software execution are deleted when it terminates, and its persistent modifications are rolled back if it was aborted.

3.6 HYBRID ATTESTATION

Hybrid attestation [52, 84, 57, 28] aims at providing the same strong security guarantees of hardware-based attestation while minimizing the required hardware. Examples of such hardware requirements include: a Read Only Memory (ROM) this is used for ensuring code and data integrity, and a Memory Protection Unit (MPU) that is required to preserve the secrecy of cryptographic keys. These requirements make hybrid attestation solutions applicable to low-end but not legacy computing devices.

3.6.1 *Minimalist Approach for Attestation*

PoSE. PoSE [109] aims at undoing malicious modifications to a device's software rather than detecting this modification. In particular, PoSE resets a device's memory to a known good state by first filling it with randomness. This randomness is then queried before installing benign software on the device. This procedure guarantees the deletion of all code from device's memory before reset. PoSE requires only a small piece of ROM for storing its code. Similar to software-based attestation, PoSE assumes adversarial silence during the protocol execution.

SMART. SMART [52] is a security architecture for low-end embedded devices, which aims at providing secure remote attestation with minimal hardware requirements. SMART's main hardware components are a ROM that stores the attestation code and a secret authentication key, and an MPU that control access to this key. ROM and MPU are inexpensive and easy to realize in hardware. They are considered minimal requirements for securing remote attestation. The security of SMART is based on the immutability of ROM, which preserves the integrity of the attestation code and the authentication key. On the other hand, MPU preserves confidentiality of the secret key by restricting access to it to attestation code only. MPU's access control is based on the value of the program counter. In particular, MPU checks whether program counter is within the address space of ROM before granting access to the key. Otherwise, the device is reset. Consequently, SMART guarantees that genuine attestation code is the only entity that has access to the secret key and is capable of generating a correctly authenticated software measurement.

Systematic Treatment. Francillon et al. present a systematic analysis of remote attestation on low-end embedded devices in order to extract its desired properties and security requirements [56, 57]. The extracted requirements are then mapped into architectural

features that are translated into hardware components. The minimal hardware requirements for securing remote attestation are identified to be a ROM, secure storage, secure memory erasure and reset mechanisms, and instructions that allow invoking attestation code from start and disabling interrupts. These requirements are claimed by the authors to be both sufficient and necessary for enabling secure remote attestation. The authors also highlight weaknesses in existing hybrid attestation solutions (e.g., SMART [52]) and devise a generic hybrid attestation scheme that satisfies the extracted properties..

TrustLite. TrustLite [84] is a lightweight security architecture that provides strong isolation on low-end embedded devices. TrustLite is based on Intel's Siskiyou Peak research platform [112]. It allows isolation of small code chunks denoted by trustlets from the rest of the system including the operating system. The main component of TrustLite is an Execution Aware Memory Protection Unit (EA-MPU), which ensures that data stored on a device is only accessible to the trustlet that owns this data. Similar to SMART's MPU, EA-MPU access control is based on the value of the program counter. In fact, TrustLite can be viewed as a generalization of SMART that: allows MPU rules to be dynamically programmed thus enabling isolation; uses secure boot to ensure authenticity of trustlet's code and data; and allows interrupts. Interrupts in TrustLite are securely handled via storing a trustlet's state in predefined CPU registers and erasing it before passing control to the legacy interrupt handler.

TyTAN. Based on TrustLite, TyTAN [28] aims at providing isolation while satisfying real-time requirements, i.e., it ensures acting reliable within strict time constraints. Real-time requirement is not considered in most of security architectures for embedded devices (including TrustLite) despite being increasingly relevant. Unlike TrustLite, TyTAN also allows dynamic loading, unloading, launching, and halting of executing trustlets at runtime. This feature increases the efficiency of TyTAN considerably by allowing better utilization of resources. Moreover, trustlets can be securely updated in order to account for software bugs. TyTAN allows local attestation where different trustlets verify each others integrity, e.g., to establish a secure communication, and remote attestation allowing a trustlet to prove its integrity to a remote entity. The security properties of TyTAN are achieved based on a small amount of software TCB which excludes the operating system. The integrity of this software TCB is ensured through means of secure boot.

3.6.2 *Attestation based on Isolation*

SPM. SPM [138] refers to software modules that are capable of isolating themselves from every other software on the same device. Similar to SMART and TrustLite, the core component of SPM is memory access control that is based on the value of the program counter. Security guarantees provided by SPM include confidentiality of data processed and stored by isolated software modules. The hardware features required by SPM are access control rules that provide confidentiality of data, and three new instructions that configure access control rules and control a software's life cycle. In addition to the hardware, SPM's TCB include a small part of the boot process and the code of the isolated module.

Sancus. Sancus [101] is security architecture for low-end embedded devices that aims at providing isolation, remote attestation, and secure communication, without relying on any software in its TCB. Isolation in Sancus is based on memory access control of SPM. This is extended with a key management protocol in order to allow secure communication and remote attestation. In particular, each software module in Sancus has its own key that is derived from the module's code, size, position in memory. Sancus leverages a new instruction that, when invoked, generates a Message Authentication Code (MAC) based on the key of the module that called this instruction. In order to avoid leakage of a module's key, MAC generation instruction is only available when memory protection is enabled for the calling module. Remote attestation in Sancus is achieved by simply generating a MAC on the challenge sent from the verifier. Sancus incurs a very high hardware cost, as it requires three additional instructions to implement its features.

3.7 RUNTIME ATTESTATION

While static attestation ensures that the code loaded for execution is not maliciously modified, runtime attestation [46, 20, 83, 65, 155, 9, 49] aims at verifying the integrity of the code's behavior at runtime. Runtime attestation usually target attacks that hijack the control flow of a program to execute an arbitrary malicious functionality without changing the underlying program code [141]. The most prominent example of runtime attacks is the Return-Oriented Programming (ROP) attack [127] that exploit a memory corruption vulnerability to stitch together small code snippets and execute a malicious functionality. Existing runtime attestation solutions measure critical aspects of a program's execution that are relevant to runtime attacks such as the base pointer of called functions. Recent effort has lead to Control-Flow Attestation (CFA) schemes that allow measuring the exact execution path followed by a program.

3.7.1 *Dynamic Attestation*

Semantic Remote Attestation. Semantic remote attestation [65] (TrustedVM) aims at enabling attestation of dynamic system properties using language-based virtual machines. The main idea of TrustedVM is to allow the enforcement of various high-level and dynamic properties on an application running within a remote party. In particular, an application is executing within a language-based virtual machine that can derive and enforce various properties in that application. Such properties include dynamic properties such as constraints on its input and runtime state, and system properties identified using test suites.

ReDAS. Dynamic remote attestation [83] (ReDAS) is an attestation scheme that aims at allowing integrity verification of the runtime behavior of the software running on a remote prover. The main idea of ReDAS is to identify a set of properties that must be satisfied by dynamic objects during their lifetimes, and use these properties to verify the overall integrity of an executing program at runtime. These properties are referred to as dynamic system properties. One example of dynamic objects considered by ReDAS is

the frame pointer that has the property of being linked to other frame pointers within the stack. ReDAS is based on identifying the challenges for dynamic attestation, which are characterized by the large number of dynamic objects whose known good states is not easy to identify, in addition to the stronger adversary that is capable of launching runtime attacks on the attestation itself. To address these challenges, ReDAS identifies two dynamic properties that are used to prove the integrity of a running program: structural and global data integrity. Finally, ReDAS allows automatic extraction of dynamic system properties for an application through the Daikon system for dynamic detection of likely invariants [53]. Further, it uses TPM to protect the integrity of the attestation report. ReDAS is not a complete solution. It is based on a small number of dynamic properties and is only capable of detecting runtime attacks that violate these properties.

Trusted Virtual Containers. Trusted virtual containers [20] (TVC) addresses the static nature of hardware-based attestation by extending attestation to a program's runtime properties. TVC identifies the shortcomings of TPM-based attestation to be scalability, dynamicity, and flexibility of trusted environments. To address these limitations, TVC leverages Solaris containers to provide virtualized environments each having a certified key pair rooted within the TPM. Containers are created on demand based on a set of trust attributes agreed upon from both the prover and the verifier. Finally, the DTrace tool of Solaris is used to monitor the behavior of programs executing within different containers and revoke the certificate of those containers that violate a set of predefined specifications.

DynIMA. The goal of DynIMA [46] is to detect ROP attacks by monitoring the behavior of a program during execution. This goal is achieved by modifying the program loader of the operating system to allow the instrumentation of all executed programs. Binary instrumentation enables monitoring dynamic events that can be either generic or specific to the executing program. Consequently, a ROP attack is detected and reported when a predefined policy is violated, e.g., three consecutive returns with less than five instructions inbetween. This solutions is based on heuristics that can cause a large number of false positive and false negatives. It has a significant performance overhead. And, it only targets ROP attacks. DynIMA is incapable of detecting other runtime attacks, e.g., Data-Oriented Programming (DOP) attacks [71].

3.7.2 *Control-Flow Attestation (CFA)*

C-FLAT. C-FLAT [9] is the first CFA scheme for detecting control-flow attacks on low-end embedded devices. It allows a device to compute a measurement of the exact execution path of an executing program and report it to a remote entity. The reported measurements enables tracing the device's execution and detecting runtime attacks that modify the control flow of the program. Security of C-FLAT is based on a hardware trust anchor that allows isolated execution, e.g., TrustLite. Moreover, it requires binary instrumentation of the attested program that is assumed not to be maliciously modified. Furthermore, C-FLAT assumes Data Execution Prevention (DEP) that thwarts malicious code injection attacks. C-FLAT has high overhead on the prover that is required to in-

terrupt and record every control-flow event, and on the verifier that should store and search of very large database of benign attestation responses. In practice, C-FLAT's applicability is limited to the attestation of very small programs.

LO-FAT. LO-FAT [49] is a hardware-based architecture for CFA that aims at overcoming the limitation of C-FLAT in terms of runtime overhead and applicability to legacy code. In particular, LO-FAT leverages custom hardware to measure a program's control flow in parallel to its execution, thus incurring no runtime overhead on the prover. LO-FAT's hardware is tightly integrated within the prover's processor, which allows it to collect information about control-flow events that are available to the processor. Therefore, it requires no instrumentation of the attested software and is applicable to legacy code. Although LO-FAT requires no new CPU instructions, it incurs a very large area overhead which hinders its adaptation in real systems.

ATRIUM. ATRIUM [155] is another hardware-based architecture that combines static and CFA. The goal of ATRIUM is to thwart a hardware adversary that can modify the program binaries at runtime, thus executing a Time-of-Check-Time-of-Use (TOCTOU) attack. Such TOCTOU attacks may allow bypassing both static attestation (e.g., SMART [52]) and CFA (e.g., C-FLAT [9]). The goal of ATRIUM is achieved by generating an attestation response that incorporates both control-flow events and program binaries measured at the time of execution. ATRIUM requires no new CPU instructions. However, its large area overhead also hinders its adaptation in real systems.

LiteHAX. LiteHAX [48] aims at detecting data-only runtime attacks that do not change the control flow of a program through means of hardware-assisted CFA. LiteHAX achieves its goal by incorporating in the attestation response a compact encoding of the execution path and a digest of memory access operations, e.g., load and store instructions. In addition to verifying the integrity of the execution path, the verifier leverages symbolic execution based on this path to generate and verify executed memory operations, thus, verifying data-flow integrity. Although LiteHAX requires minimal non-invasive interfacing with the processor with no new instructions introduced, it is unlikely to be realized in real systems due to its large area overhead.

3.8 COMPARISON

In Table 3.1 we compare existing attestation classes in terms of properties, requirements, and the attacks they detect or are vulnerable to. The basic information that can be extracted from the table are summarized in the following:

- Software-based attestation has no hardware or cryptographic requirements. However, it provides weak security guarantees as its based on strong assumptions (e.g., adversarial silence during attestation). This has made software-based attestation the target of a wide range of attacks (e.g., proxy/collusion attack). Furthermore, software-based attestation induces a very high overhead on the prover that is required to randomly access its whole memory, and the verifier that is expected to simulate the prover's execution.

Table 3.1: Comparison between different classes of attestation

		Attacks											Requirements							Properties							
													Hardware			Software			Crypto	Efficiency				Security			
		Communication	**Malware**	**Physical**	**Run-time**	**Collusion**	**Code Hiding**	**Address Translation**	**Page Substitution**	**Code Compression**	**Memory Shadowing**	**TOCTOU**	**Complex**	**Simple**	**Other**	**Source Code**	**Instrumentation**	**Optimality**	**Secret Keys**	**Key Management**	**Prover's Efficiency**	**Verifier's Efficiency**	**Scalability**	**Time Consistency**	**Hardware Independence**	**End-to-End Security**	**Resiliency to TOCTOU**	**Detection at Run-time**
Software	Timed	✘	✔	✘	✘	✘	✘	✘	✘	○	✘	✘	○	○	○	○	○	✔	○	○	○	○	○	○	○	○	○	○
	Others	✘	✔	✘	✘	✘	○	○	○	✘	✘	✘	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
Hardware	TPM	○	✔	✘	✘	○	○	○	○	○	○	✘	✔	○	○	○	○	○	✔	✔	✔	✔	○	○	○	✔	○	○
	Others	○	✔	✘	✘	○	○	○	○	○	○	✘	✔	○	○	○	○	○	✔	✔	✔	✔	○	○	○	✔	○	○
Hybrid	Minimalist	○	✔	✘	✘	○	○	○	○	○	○	✘	○	✔	○	○	○	○	✔	✔	✔	✔	○	○	○	✔	○	○
	Isolation	○	✔	✘	✘	○	○	○	○	○	○	✘	○	○	✔	○	○	○	✔	✔	✔	✔	○	○	○	✔	○	○
Run-time	Control-flow	○	○	✘	✔	○	○	○	○	○	○	✘	○	✔	○	✔	✔	○	○	✔	✔✢	○	○	○	○	✔	○	✔
	Dynamic	○	○	✘	✔	○	○	○	○	○	○	✘	○	✔	○	✔	✔	○	○	✔	○	○	○	○	○	✔	○	✔

✔ — detects/requires/provides ✘ — is vulnerable to ○ — is not relevant
✢ C-FLAT [9] does not satisfy this property.

- Hardware based attestation provides better efficiency and stronger security guarantees. This has made hardware-based attestation resilient to a wide range of attacks that target software-based attestation. However, hardware-based attestation requires complex and expensive hardware and is not applicable to low-end and legacy computing devices. Moreover, Hardware-based attestation is vulnerable to a number of attacks, e.g., physical and runtime attacks.
- The security guarantees and efficiency of hybrid attestation are similar to that provided by hardware-based attestation. This is achieved with simpler hardware requirements. As a consequence, hybrid attestation currently presents the best alternative for enabling attestation on low-end embedded devices.

- Runtime attestation allows the detection of more sophisticated runtime attacks that do not change the program code, e.g., control-flow attacks detected by Control-Flow Attestation (CFA). However, the requirements imposed by existing CFA schemes may not be applicable in all realistic settings, e.g., access to source code of the attested program. Furthermore, they induce a high overhead on the prover that is required to measure every single control-flow event, and the verifier that is expected to search a very large database of benign control-flow paths.

Finally, its important to mention that static attestation for a single device seems to be a well established area research. However, runtime attestation solutions could still be enhanced in order to provide better efficiency and allow the detection of a wider range of runtime attacks, e.g., Data-Oriented Programming (DOP) attacks [71]. On the other hand, both static and dynamic attestation solutions cannot be applied to emerging systems that are formed of a very large number of embedded devices. In particular, due to reasons of efficiency, these solutions cannot be scaled to attest a large number of devices. Furthermore, they cannot be extended to large embedded networks as they are vulnerable to attacks that are more relevant in these networks, e.g., physical attacks.

3.9 CONCLUSION AND OPEN PROBLEMS

Over the past two decades, attestation has become a popular security solution that allows establishing trust in a remote device. The goal of attestation has evolved from the detection of malware infestation that changes a software binaries to ensuring the correct execution of that software. Existing attestation solutions are not applicable to emerging embedded networks that are formed a very large number of devices which hinders their deployment. In fact, these solutions have high overhead that does not scale to very large systems, and are vulnerable to attacks that are relevant for such systems. Consequently, designing scalable attestation solutions that are applicable to such networks and are capable of detecting a wider range of attacks is considered a challenging open problem.

4

DETECTION OF MALWARE INFESTATION

Envisioned embedded deployments involve networks that are formed of a very large number of heterogeneous embedded devices. Such networks represent an attractive target for attacks. While the most prominent attack on embedded systems is malware infestation, e.g., Stuxnet [148], remote attestation represents the most popular mechanism for detecting malware infestation attacks. Unfortunately, all existing attestation schemes consider one single prover and one verifier. They are not scalable to large networks of embedded devices. In this chapter, we present two attestation protocols for large networks of embedded devices, i.e, we devise two collective attestation solutions that leverage in-network verification and aggregate signatures to scale static attestation to large embedded networks, thus allowing efficient detection of malware infestation in such networks. In Section 4.1, we present the first attestation protocol that is capable of efficiently and collectively verifying the software integrity of a very large number of embedded devices. The presented solution adapts the assumptions of single-device attestation in order to achieve best efficiency. It targets large embedded networks that are physically protected. Next, Section 4.2 presents another collective attestation solution for embedded networks that is applicable to a wider range of scenarios. This solution exploits our novel aggregate signature construction to achieve its goals while inducing minimal additional runtime overhead. The two presented solutions complete each other. They present a tradeoff between security and efficiency. While the solution in Section 4.1 provides better efficiency when applied to scenarios with no physical attacks, the solutions of Section 4.2 should be applied when physical attacks on some of the devices is possible inducing a limited additional overhead. Finally, we present in Section 4.3 a systematic analysis of collective attestation in terms of requirements, components, and security guarantees. Our analysis aims at putting collective attestation on solid ground and providing a guide for researchers in this area.

Remark. The results presented in this chapter are due to the author of this work and the result of many intensive discussions and collaboration with Christian Wachsmann (Intel Labs, Nürnberg), Moreno Ambrosin (Intel Labs, Oregon), Gregory Neven (IBM Research, Zurich), Matthias Schunter (Intel Labs, Darmstadt), Shaza Zeitouni (TU Darmstadt, Germany), Ghada Dessouky (TU Darmstadt, Germany), Gene Tsudik (UCI, California), Mauro Conti (University of Padua, Italy), N. Asokan (Aalto University, Finland), Ferdinand Brasser (TU Darmstadt, Germany), and Ahmad-Reza Sadeghi (TU Darmstadt, Germany). Parts of this chapter have been published in [16], [12], [77], and [102].

4.1 SCALABLE EMBEDDED DEVICE ATTESTATION

Large networks of embedded devices, such as industrial control systems, have been the target of a wide range of attacks. A prominant example of such attacks is the Stuxnet

episode [148]. A key challenge for securing embedded networks and their operations is verifying the software integrity of individual devices in the face of malware infestation attacks. This illustrates the need for an attestation solution that efficiently and collectively verifies the software integrity and correct operation of a large number of low-end embedded devices. Indeed, such a solution should overcome multiple challenges related to network discovery, key management and routing, while maintaining scalability and low overhead.

The solution we present in this section targets centrally managed systems such as IoT networks e.g., home/office automation. These networks are usually formed of heterogeneous mobile devices. The burden of the solution, in terms of communication, computation, and energy cost is distributed over all the devices in the network. This is crucial to prevent some devices, e.g., the verifier, from becoming a performance bottleneck.

Contribution. We investigate the security of large dynamic networks of heterogeneous embedded devices and design the first collective attestation solution for these networks. Our solution is generic, i.e., it is independent of the underlying software measurement process which can be either static or dynamic. It represents the first step in our line of research on collective attestation. We assess the security of our solution in a security model that allows software-only attacks assumed in most existing attestation schemes. Moreover, in order to demonstrate feasibility, we show how to instantiate the solution on two security architecture for low-end embedded devices with different security features and functional capabilities: SMART [52] and TrustLite [84]. Finally, we present extensive performance evaluation based on these two instantiations, in addition to simulations of the solution in networks of up to $1,000,000$ devices. The solution allows a verifier to verify the integrity of a network formed of one million devices in order of seconds. This is achieved by distributing the burden of attestation on all the devices in the network and enabling neighboring devices to attest each other, accumulate attestation result, and send it to the verifier. Further, in order to provide efficiency our solution uses symmetric key cryptography for authentication between devices in the network, while public key cryptography is used between the network and the verifier in order to enable public verfiability.

Outline. After providing a brief overview of our solution in Section 4.1.1, we present its details in Section 4.1.2, and describe our implementation in Section 4.1.3. Performance evaluation is then presented on Section 4.1.4. Security of the solution is examined in Section 4.1.5, possible extensions are described in Section 4.1.6, and this section concludes in Section 4.1.7.

4.1.1 *Collective Attestation*

4.1.1.1 *Problem Description and System Model*

We consider a dynamic network $\mathcal{N}$ that is formed of n interconnected heterogeneous devices. The network operator O is responsible for initializing each device D_i in a secure environment. V is an entity that is interested in assessing the trustworthiness of $\mathcal{N}$. Further, $\mathcal{N}$ may not have a routing protocol in place. However, devices in $\mathcal{N}$ should

be able to communicate to their direct neighbors [44, 68, 114, 115]. Since the mobility of devices can be involuntary, i.e., guided by ambient factors, neither operator O nor the verifier V are assumed to be aware of the network topology at any given time. A collective attestation solution allows a verifier V (which could be a local or remote entity) to verify the software integrity of $\mathcal{N}$ in an efficient and scalable manner. The basic idea is that $\mathcal{N}$ is considered trustworthy if all of its devices have a benign software deployed and certified by O, i.e., $\mathcal{N}$ is trustworthy if none of its devices is infested by malware.

4.1.1.2 *Requirements Analysis*

Objectives. A collective attestation solution for a network of embedded devices should satisfy the following five properties:

- *Property #1:* Allow the attestation of the network $\mathcal{N}$ as a whole.
- *Property #2:* Be efficient and scalable, i.e., more efficient than attesting every device in $\mathcal{N}$ individually.
- *Property #3:* Not depend on V's knowledge of $\mathcal{N}$'s configuration, e.g., topology or types or version of software on devices.
- *Property #4:* Allow parallel execution of multiple attestation instances, e.g., by different verifiers.
- *Property #5:* Not depend on the properties of the underlying measurement scheme used by devices in $\mathcal{N}$.

The main objective of collective attestation is satisfying property #1 and property #2, which are critical in large networks. Further, collective attestation is simplified by property #3 which is required when V has no access to device's software configuration, e.g., outsourced maintenance in smart factories which requires setup of production system to remain secret [98, 90, 158]. Property #4 is needed when multiple verifiers are expected to independently attest the network. Finally, property #5 enables extensibility by supporting applicability to wide range of measurement mechanisms, e.g., static or Control-Flow Attestation (CFA).

Adversary Model. We assume an adversary $\mathcal{A}$ that can modify the software on any device D_i in $\mathcal{N}$ except what is protected by hardware, e.g., $\mathcal{A}$ cannot modify code stored in ROM. $\mathcal{A}$ can eavesdrop on, modify, replay, or drop any message exchanged between devices in $\mathcal{N}$ and between any device D_i and the verifier V. We assume that both the operator O and V are trusted. Moreover, $\mathcal{A}$ is not capable of tampering with the hardware of any device D_i, i.e., physical attacks are ruled out in the present context. A software-only adversary is a common adversary model in most existing attestation schemes. This adversary is realistic in most existing IoT networks, e.g., smart home or office, and is in line with our goal of detecting malware infestation. Furthermore, we assume a stealthy adversary that aims at compromising as much devices as possible while evading detection by our solution, hence, we do not consider Denial of Service (DoS) attacks on the attestation protocol itself which would reveal $\mathcal{A}$'s presence.

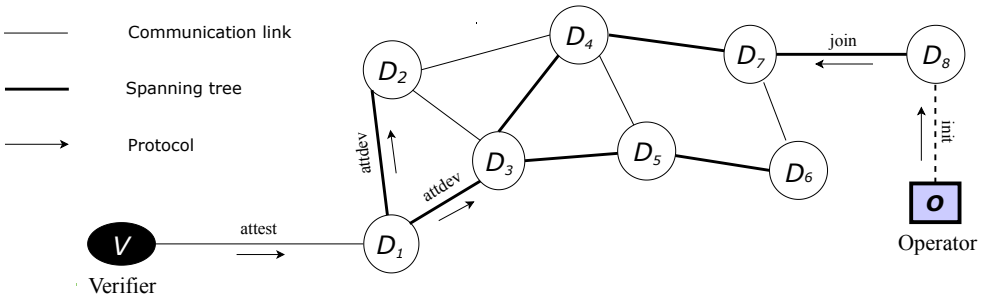


Figure 4.1: Example 8-device network: $D_1, \ldots, D_8$

Device Requirements. The design of our solution satisfies property #4 and property #5. However, satisfying the remaining three properties requires the ability to remotely attest each device in $\mathcal{N}$. Consequently, the following requirements should be satisfied by each D_i in $\mathcal{N}$ [52, 57]:

- *Integrity measurement:* $\mathcal{A}$ should not be able to tamper with the measurement mechanism responsible for measuring the software state of any device D_i.
- *Integrity reporting:* $\mathcal{A}$ should not be able to forge the measurement c_i' of D_i's software that is sent to V.
- *Secure storage:* $\mathcal{A}$ should not have access to the cryptographic key(s) that are used in the attestation protocol.

If one of these requirements is not achieved, V might not be able to detect whether $\mathcal{A}$ has tampered with D_i's software. The implementation of our solution in Section 4.1.3 exploits two security architecture for low-end embedded devices that satisfy these requirements (SMART [52] and TrustLite [84]).

Assumptions. Devices in $\mathcal{N}$ may be heterogeneous (i.e., have multiple hardware and software configurations). However, we assume that every device D_i in $\mathcal{N}$ satisfies the above requirements that allow secure remote attestation. Devices can communicate with each other, i.e., every device D_i can at least communicate to its neighboring device. Further, during the execution of the attestation protocol, the network is assumed to be connected and its topology should remain static. In particular, each device D_i in $\mathcal{N}$ should be reachable. Cryptographic primitives are assumed to be secure along with their implementations.

Protocol Overview. Figure 4.1 shows an overview of our solution in a network formed of eight devices D_1 through D_8. The proposed solution consists of three protocols: device initialization, device registration, and attestation. In the *device initialization* protocol init, O initializes each device (D_8 in Figure 4.1) in a secure environment with cryptographic secrets that are later used in the device registration and attestation protocols. When a device (e.g., D_8) detects a new neighbor (e.g., when it is introduced into, or changes its

position in, $\mathcal{N}$), it runs the *device registration* protocol join to establish attestation keys and learn the software configuration of new neighbors. Keys and software configurations are required for executing the attestation protocol. In the *attestation* protocol, V verifies the software integrity of all devices in $\mathcal{N}$. The attestation protocol consists of three sub-protocols: (1) attest executed between V and $\mathcal{N}$ (i.e., between V and D_1), (2) attdev executed between devices in $\mathcal{N}$ (e.g., between D_1 and D_2), (3) clear which ends the attestation session between V and $\mathcal{N}$. attest is based on pubic key cryptography to support arbitrary verifiers that may be unknown to O at deployment time. These verifiers do not share a keys with any device in $\mathcal{N}$. attdev is based on symmetric cryptography in order to minimize cryptographic costs. Attestation begins with V contacting a random device D_1 denoted by initiator, i.e., initiating attest. From that point on, each device in $\mathcal{N}$ first attests all of its direct neighbors (in parallel) and then returns the accumulated attestation result to the device that sent the attestation request, i.e., devices recursively perform attdev.

4.1.2 *Protocol Description*

In this section we describe the details of the protocols involved in our solution. We present the protocols in two different scenarios: Section 4.1.2.1 describes the scenario where attestation is realized as an interactive protocols, and Section 4.1.2.2 describes the non-interactive scenario.

4.1.2.1 *Interactive Protocol Description*

All existing attestation schemes are realized as an interactive protocol initiated by the verifier through sending the prover a random challenge. The challenge reflects the verifier's interest in checking the trustworthiness of the prover. It also serves as a mean for mitigating replay attacks. This setting seems natural when attestation is driven by the verifier's desire to attest the prover. However, it renders the prover susceptible to Denial of Service (DoS) attacks. When attestation is executed interactively, the details of protocols involved in our collective attestation solution are as follows:

Device Initialization. Before deployment, the network operator O initializes each device D_i in $\mathcal{N}$ with O's public verification key, which is needed for verifying certificates issued by O, a software configuration c_i (e.g., the hash of D_i's binaries), a signing key pair $(sk_i, pk_i) \leftarrow \mathsf{genkeysign}(1^\ell)$, and two certificates issued and signed by O: An identity certificate $\mathsf{cert}(pk_i)$ certifying that pk_i is D_i's public key to which it securely store a secret key sk_i, and software configuration certificate $\mathsf{cert}(c_i)$ certifying that D_i's reference software configuration is c_i. Finally, D_i's list of active session identifiers is also initialized $\Omega_i \leftarrow \{\}$. Initialization is done in a secure environment and is formally:

$$\mathsf{init}(c_i, 1^\ell) \rightarrow \left(sk_i, pk_i, \mathsf{cert}(pk_i), \mathsf{cert}(c_i), \Omega_i\right).$$

Device Registration. When a device D_i detects a new neighbor D_j (e.g., when D_i changes its position in $\mathcal{N}$ or joins another network), it runs the join protocol to es-

tablish an attestation key k_{ij} with D_j and learn D_j's reference software configuration. One approach to establish these keys is using an authenticated key agreement protocol based on signing keys sk_i and sk_j and identity certificates $\text{cert}(pk_i)$ and $\text{cert}(pk_j)$ (e.g., authenticated Diffie Hellman). Key establishment can also be done using key predistribution [30]. Through join, each device D_i shares one attestation key k_{ij} with each of its neighbors. We denote the set of all attestation keys of D_i by $\mathcal{K}_i$.

In addition sharing keys, D_i and D_j exchange their code certificates $\text{cert}(c_i)$ and $\text{cert}(c_j)$. D_i and D_j then verify these certificates using O's public key. If verification succeeds, D_i and D_j store c_j and c_i respectively. Otherwise, if D_i cannot verify the software configuration of D_j it does not accept D_j as a neighbor. Note that, this solution does not hinder software updates. a device D_i whose software is updated should only send the new software configuration certificate to its current neighbors for verification. join is formally:

$$\text{join}\left[D_i : sk_i; D_j : sk_j; * : \text{cert}(pk_i), \text{cert}(pk_j), \text{cert}(c_i), \text{cert}(c_j)\right] \rightarrow \left[D_i : k_{ij}; D_j : k_{ij}\right].$$

Device Attestation. Our solution internally uses the attdev protocol (see Figure 4.2), where a device D_i verifies the integrity of the software configuration of another device D_j. A session identifier q is used to identify different instances of the attestation protocol. The goal of q is to allow multiple protocol instances to run concurrently and to enable the construction of an attestation *spanning tree* over $\mathcal{N}$. In particular, when a new q is received by D_j from D_i, D_j marks D_i as its parent in the spanning tree and stores q as an active session identifier in the list $\mathcal{Q}_j$. Consequently, a spanning tree is formed, where connected nodes are neighbors in $\mathcal{N}$. The construction of the spanning tree can be influenced by setting up an upper bound for the number of children for each node. This allows the spanning tree grow in height while limiting its fan-out, thus leading to the formation of balanced trees from mesh networks. We leverage this measure to enable optimizing our attestation solution (see Section 4.1.4). In attdev, each device D_j is responsible for attesting its children in the tree. The results of this attestation are then accumulated along with the results reported to these children by their children, i.e., attestation results for the subtrees rooted at each of the children. The accumulated result is then reported by D_j to its parent D_i. attdev is executed in parallel with every neighbor D_k of D_j, using only symmetric cryptography. It is based on the attestation key k_{ij} and the reference software configuration c_j established through join.

The output of attdev for a device D_i is: The bit b that indicates whether the attestation of child D_j was successful, i.e. it returns $b = 1$ if the software configuration of D_j matches the software configuration stored at D_i and $b = 0$ otherwise. The number β of devices whose authenticity and software integrity has been successfully verified in the subtree rooted at D_j (excluding D_j). And the total number τ of attested devices in the subtree rooted at D_j (excluding D_j).

If a device D_j received an old global session identifier for an attestation instance to which it has already participated, D_j does not respond to that session identifier. The

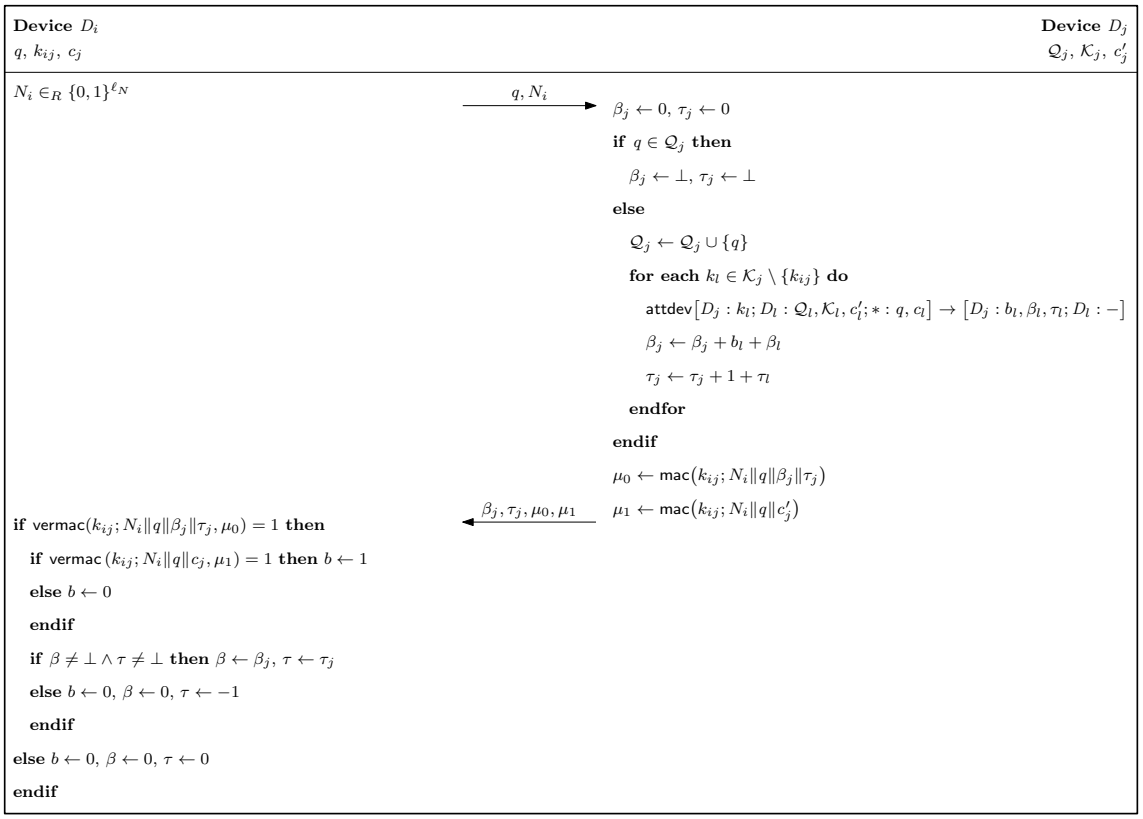


Figure 4.2: Protocol attdev

output of attdev for D_i is then $b = 0$, $\beta = 0$, and $\tau = -1$ after a time-out has occurred. attdev is formally:

$$\mathsf{attdev}\left[D_i : k_{ij}; D_j : \mathcal{Q}_j, \mathcal{K}_j, c'_j; * : q, c_j\right] \rightarrow \left[D_i : b, \beta, \tau; D_j : -\right].$$

The details of attdev are shown in Figure 4.2 and are as follows: D_i sends each neighbor D_j a fresh nonce N_i and a session identifier q. D_j then checks whether q is an active session identifier belonging to its list $\mathcal{Q}_j$. If so, it replies with $\beta_j \leftarrow \perp$ and $\tau_j \leftarrow \perp$. Otherwise, if q is a new session identifier, D_j executes attdev with its every neighbor D_k and adds q to the list $\mathcal{Q}_j$. Eventually, when D_j receives the attestation results from all neighbors, it accumulates them into β_j and τ_j, which are then authenticated with a MAC μ_0 based on k_{ij}. D_j also attests itself to D_i with the MAC μ_1 over its current software configuration. If μ_0 and μ_1 verify successfully, D_i accepts the attestation response of D_j.

Network Attestation. To verify the integrity of $\mathcal{N}$, V initiates attest by contacting an arbitrary $D_1 \in \mathcal{N}$. This initiates a recursive attestation process in $\mathcal{N}$. Eventually, V returns a bit $b = 1$ indicating that the attestation of $\mathcal{N}$ was successful, or $b = 0$ if the attestation failed. D_1 is either chosen randomly by V or based on preference or location. Note that,

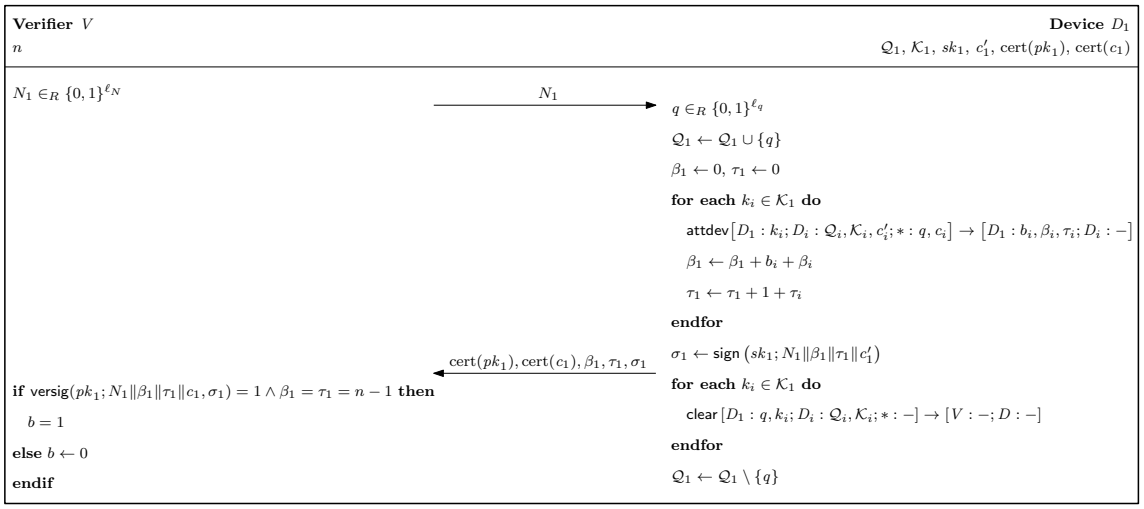


Figure 4.3: Protocol attest

V can either be remote or within the communication range of some devices in $\mathcal{N}$. attest is formally:

$$\text{attest}\left[V : -; D_1 : \mathcal{Q}_1, \mathcal{K}_1, sk_1, c_1'; * : n, \text{cert}(pk_1), \text{cert}(c_1)\right] \rightarrow [V : b; D_1 : -]\,.$$

The attest protocol is shown in Figure 4.3 and works as follows: The protocol starts by V sending an attestation request to D_1 that contains a fresh nonce N_1. D_1 then generates a new session identifier q and sends it to all its neighbors along with a a new fresh N_i for each neighbor D_i, i.e., D_1 executes attdev with each of its neighbors. The neighbors then recursively execute attdev with their neighbors. While q identifies the current attestation instance and assist in the construction of the spanning tree, the goal of the nonces is to thwart replay attacks on the communication between devices and between D_1 and V. Eventually, accumulated attestation reports of all the devices in $\mathcal{N}$ are received by D_1, which accumulates them into τ_1 and β_1. D_1 then authenticates τ_1, β_1, and its current software configuration c_1' with a signature σ_1 based on its secret signing key sk_1. It then sends σ_1, β_1, and τ_1 along with its certificates cert(pk_1) and cert(c_1) to V. Using O's public key, cert(pk_1) and cert(c_1), V first verifies the authenticity of pk_1 and c_1. V then uses pk_1, c_1 and σ_1 to verify the authenticity of β_1, τ_1. If verification of σ_1 succeeds, V concludes that attestation was successful. V then uses τ_1 and β_1 to learn the respective number of devices that participated in protocol and were successfully attested, and to determine the output b of attest. Attestation fails if the verification of σ_1 fails or D_1 did not respond to V. When D_1 sends its response to V, it also initiates the clear protocol in order to delete the current global session identifier q from every device in $\mathcal{N}$.

Clear. A device D_i sends to every neighbor D_j a global session identifier that is authenticated with k_{ij}. Upon receiving q, each neighbor D_j deletes q from its list of active session identifiers $\mathcal{Q}_j$, and executes clear recursively with all its neighbors. clear is formally:

$$\text{clear}\left[D_i : q, k_{ij}; D_j : \mathcal{Q}_j, \mathcal{K}_j; * : -\right] \rightarrow \left[D_i : -; D_j : -\right].$$

4.1.2.2 *Non-Interactive Protocol Description*

On the other hand, there exists certain scenarios where attestation is triggered by factors that can be known to the prover, e.g., regular maintenance. In such scenarios, attestation can also be realized as a non-interactive protocol initiated by the prover. When attestation is executed non-interactively, the details of the protocols involved in our collective attestation solution deviate from the protocol description presented above. Moreover, in addition to the devices requirements in Section 4.1.1, non-interactive attestation requires every device D_i in $\mathcal{N}$ to be equipped with: (1) a Reliable Read-Only Clock (RROC) that cannot be modified by any software that is residing on D_i, and (2) a dedicated timeout circuit that allows triggering attestation without revealing the attestation time. Finally, non-interactive collective attestation assumes the existence of a spanning tree over $\mathcal{N}$, and time synchronization between V and every device in $\mathcal{N}$. In the following we present details of the protocols involved in our solution when attestation is non-interactive:

Device Initialization. In addition to key pairs, software configuration, and certificates, every device D_i in $\mathcal{N}$ is also initialized by the network operator O with a random seed s_i that is required to randomly generate attestation time. Initialization is formally:

$$\mathsf{init}(c_i, 1^\ell) \rightarrow \left(sk_i, pk_i, \mathsf{cert}(pk_i), \mathsf{cert}(c_i), s_i\right).$$

Device Attestation. The non-interactive solution uses attdev protocol to allow a device D_i to verify the integrity of its children in the spanning tree. This protocol is similar to the attdev protocol described above. However, it is initiated by the child nodes at the attestation time t_{attest}. Further, it uses t_{attest} instead of fresh nonces to protect against replay attacks. In particular, at t_{attest}, every device D_j sends to its parent D_i in the spanning tree an attestation response including: the number β_j of devices whose authenticity and software integrity has been successfully verified in the subtree rooted at D_j (excluding D_j); the total number τ_j of attested devices in the subtree rooted at D_j (excluding D_j); a MAC μ_0 over τ_j, β_j, and t_{attest}; and a MAC μ_1 over D_j's software configuration c'_j and t_{attest}. The parent D_i verifies the MACs, accumulates the results received from its children, and reports the accumulated result to its parent in the tree. The output of attdev for a device D_i is: The bit b that indicates whether the attestation of child D_j was successful, and the numbers β and τ received from D_j. Protocol attdev is formally:

$$\mathsf{attdev}\left[D_i : k_{ij}, s_i; D_j : \mathcal{K}_j, c'_j, s_j; * : c_j\right] \rightarrow \left[D_i : b, \beta, \tau; D_j : -\right].$$

After attesting itself to its parent, each device D_j uses the random seed s_j to determine the next attestation time, i.e., $t_{\mathsf{attest}} = t_{\mathsf{attest}} + \mathsf{rand}(s_j)$, where rand is a Pseudorandom Number Generator (PRNG).

Network Attestation. The protocol attest allows the network to attest itself to the verifier V. This protocol is similar to the attest protocol described above. However, it is initiated at t_{attest} by the root device D_1 of the spanning tree. Further, it also uses t_{attest} instead of fresh nonces to protect against replay attacks. In particular, at t_{attest}, D_1 accumulates all the attestation responses coming from its children along with its own software configuration into a single response that is formed of: the number β_1 of devices in $\mathcal{N}$ whose

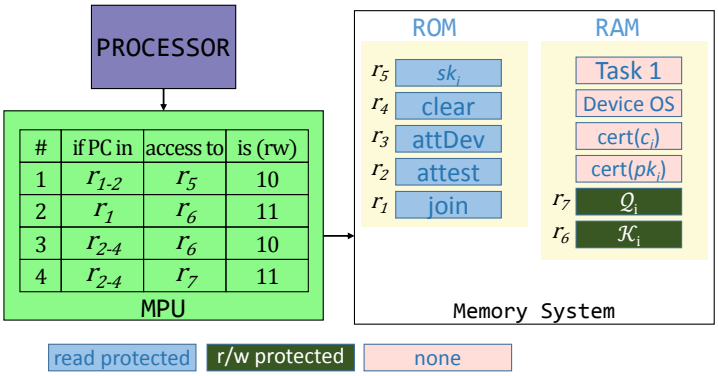


Figure 4.4: Implementation based on SMART [52]

authenticity and software integrity has been successfully verified (excluding D_1); the total number τ_1 of attested devices in $\mathcal{N}$ (excluding D_1); and a digital signature σ_1 over τ_j, β_j, D_1's software configuration c'_1, and t_{attest}. D_1 then sends σ_1, β_1, and τ_1 along with its certificates $\text{cert}(pk_1)$ and $\text{cert}(c_1)$ to V. V accepts the attestation response only if received within short interval around t_{attest}, i.e., within $[t_{\text{attest}} - \delta_t, t_{\text{attest}} + \delta_t, +t_{tr}]$ where δ_t is the maximum clock skew between V and $\mathcal{N}$ and t_{tr} is the upper bound on the time required to perform collective attestation. After verifying σ_1, β_1, and τ_1, V determines the output b of attest as above. Protocol attest is formally:

$$\text{attest}\left[V : s_1; D_1 : \mathcal{K}_1, sk_1, c'_1, s_1; * : n, \text{cert}(pk_1), \text{cert}(c_1)\right] \rightarrow [V : b; D_1 : -]\,.$$

4.1.3 *Implementation*

We present two instantiations of our solution, based on interactive attestation, on top of two lightweight security architectures for low-end embedded systems: SMART [52] and TrustLite [84] (see Chapter 3). The two architectures we chose provide strong security guarantees for remote attestation while imposing minimal hardware features, i.e., a small amount of Read-Only Memory (ROM) and a simple Memory Protection Unit (MPU).

Implementation on SMART. In order to enable secure implementation of our solution on SMART [52]. we require slight modifications to SMART's architecture allowing the MPU to control access to a small amount of rewritable memory. Rewritable memory is needed to securely store cryptographic keys and session identifiers generated during the execution of the protocols, e.g., attestation keys established through join and used in attdev. For our implementation, we store in ROM of every device the program code, i.e. the code responsible for executing the protocols join, attdev, attest, and clear. We also store in ROM of each device D_i the signing key sk_i. The integrity of the code and the signing key is then ensured through the emutability of ROM. Further, we store the lists $\mathcal{Q}_i$ of active session identifiers, and $\mathcal{K}_i$ of keys shared with neighbors in rewritable memory of each device D_i. Note that, these lists are updated during the lifetime of D_i. Our implementation is shown in Figure 4.4 where rewritable memory is denoted by

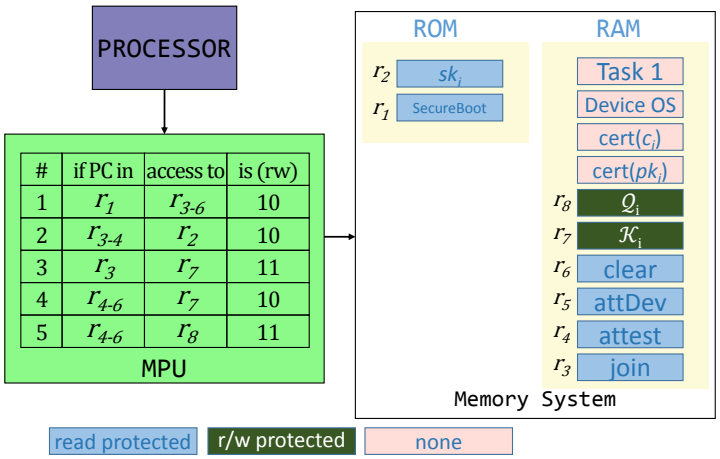


Figure 4.5: Implementation based on TrustLite [84]

RAM. SMART's MPU ensures that secret data are only accessible to unmodified protocol code in ROM that requires access to this data. For example rule #1 ensures that only join and attest have read access to sk_i, rules #2 and #3 ensure that only join, attdev, attest, and clear have read access to $\mathcal{K}_i$, which is write accessible to join only, and rule #4 ensures that only attdev, attest, and clear have write access to $\mathcal{Q}_i$, which is read accessible to all code on D_i.

Implementation on TrustLite. Our solution is implemented as trustlets on the TrustLite security architecture [84] (see Chapter 3). More precisely, we implemented each of the protocols join, attdev, attest, and clear as a single independent trustlet on device D_i. Our implementation is shown in Figure 4.5. Trustlite ensures the software integrity of each of these protocols via the secure boot component SecureBoot on D_i. Further, as in SMART, the MPU of TrustLite ensures that secret data of D_i is only accessible to appropriate trustlets. For example rule #1 ensures that SecureBoot has exclusive read access to the memory housing the program code of join, attdev, attest, and clear, rule #2 ensures that only join and attest have read access to sk_i, rules #3 and #4 ensure that join, attdev, attest, and clear have exclusive read access to $\mathcal{K}_i$, which is write accessible to join only, and rule #5 ensures that only attdev, attest, and clear have write access to $\mathcal{Q}_i$, which is publicly read accessible.

4.1.4 *Performance Evaluation*

We assess performance of our solution, based on interactive attestation, in terms of computational, communication, memory, runtime, and energy costs. We further present simulation results for networks of up to $1,000,000$ devices. Our performance evaluation is based on the implementation in Section 4.1.3. It assumes that the topology of the network does not change during the execution of the attestation.

Computation Cost. Cryptographic operations, such as MAC generation, constitute the major part of the computation cost. Let g_i denote the number of neighbors of every

device D_i, and $h_i \leqslant g_i - 1$ be the upper bound on the number of children of D_i in the spanning tree. The initiator D_1, which is chosen by the verifier V to be the interface to $\mathcal{N}$, verifies up to $2g_1$ MACs and generates only 1 digital signature. Each device D_i verifies up to $2h_i$ MACs and generates only 2 MACs.

Communication Cost. We used HMAC based on SHA-1 as our MAC implementation, and ECDSA as our digital signature scheme, i.e., $\ell_{\mathsf{mac}} = 160$ and $\ell_{\mathsf{sign}} = 320$. We further used a 64 bit counter for τ, and β, and chose $\ell_N = 160$ and $\ell_q = 64$. As a consequence, signing keys, nonces, and MACs, are 20 Bytes each. β, τ, and the global session identifier q are 8 Bytes each. Digital signatures are 40 Bytes each. And, certificates are 60 Bytes each. The initiator D_1 has a communication overhead, which is upper bounded by receiving $20 + 56g_1$ Bytes and sending $48g_1 + 176$ Bytes. The upper bound on the communication overhead of each device D_i is receiving $68g_i + 56$ Bytes and sending $56g_i + 68$ Bytes.

Memory Cost. Every device D_i in $\mathcal{N}$ should store the following: (1) an authentication key pair (sk_i, pk_i) and the corresponding identity certificate $\mathrm{cert}(pk_i)$; (2) a software configuration c_i and the corresponding software configuration certificate $\mathrm{cert}(c_i)$; (3) a set of attestation keys that it shares with neighboring devices – $\mathcal{K}_i$; and (4) the list $\mathcal{Q}_i$ of active session identifiers that should at least hold one active identifier q. The memory cost of D_i is around $20g_i + 168$ Bytes. Low-end embedded devices, which we target in this solution, have more than 1,024 Bytes of Flash memory. Applications where devices have 12 neighbors require each device to use less than half of this memory. Other applications that require more neighbors per device usually incorporate more power devices with larger Flash memory, e.g., vehicular ad-hoc networks.

Runtime. The design of our solution allows devices at the same level of the spanning tree to perform their execution of the attestation protocol at the same time, i.e., in parallel. However, the verification of MACs on level l of the tree requires the generation of these MACs by level $l - 1$. Consequently, runtime of the attestation protocol is dependent on the overall height $d = f(n) \in \mathcal{O}(\log(n))$ of the spanning tree.[1] Moreover, as can be seen from the computation costs per device, the overall runtime is also affected by the fan-out of the tree, i.e., the number of children per device.

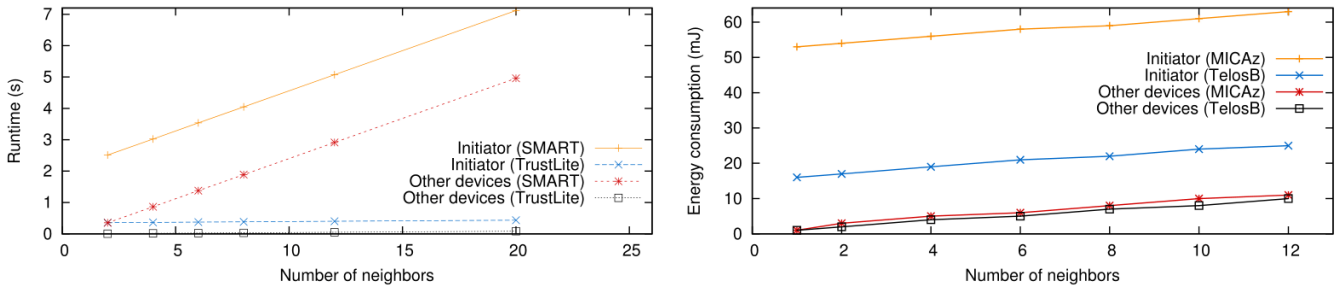
We denote by t_{sign}, t_{mac}, t_{prng}, and t_{tr} the time that a device D_i requires to perform the sign operation, to execute mac or vermac, to generate a fresh nonce, and to communicate 1 Byte to a neighboring device D_j respectively. The overall runtime of the attestation protocol is:

$$t \leqslant \left(280 + 168d + \sum_{i=0}^{d} g_i\right) t_{\mathsf{tr}} + \left(2 + 4d + \sum_{i=0}^{d} g_i\right) t_{\mathsf{mac}} + (d+1) t_{\mathsf{prng}} + t_{\mathsf{sign}}$$

We show in Figure 4.6a the runtime of the attestation protocol per device. The runtime of attestation on one device increases linearly on with the number of neighbors of this device.[2] Further, because of public key cryptography, runtime on the initiator device D_1

1 The height d of the spanning tree excludes the initiator D_1, i.e., the height of a tree formed of one device is $d = 0$.

2 Due to the graph scale, runtime on TrustLite appear to be constant when it is in fact linear in number of neighbors.



(a) Runtime of attestation

(b) Energy consumption of attestation

Figure 4.6: Performance of attestation per device

is more than that on any other device in $\mathcal{N}$. Finally, TrustLite implementation seems more efficient than that of SMART because of the higher clock speed of TrustLite hardware.

Energy Cost. We denote by E_{prng}, E_{sign}, E_{mac}, E_{recv}, E_{send}, the energy that a device D_i needs for generating a fresh nonce, performing the sign operation, executing mac or vermac, receiving 1 Byte, and sending 1 Byte respectively. During one execution of the attestation protocol, the energy E_1 consumed by initiator D_1 can be estimated as:

$$E_1 \leqslant (176 + 68g_1)E_{send} + (20 + 56g_1)E_{recv} + 3g_1 E_{mac} + g_1 E_{prng} + E_{sign}$$

Further, the energy E_i consumed by every other device D_i in $\mathcal{N}$ can be estimated as:

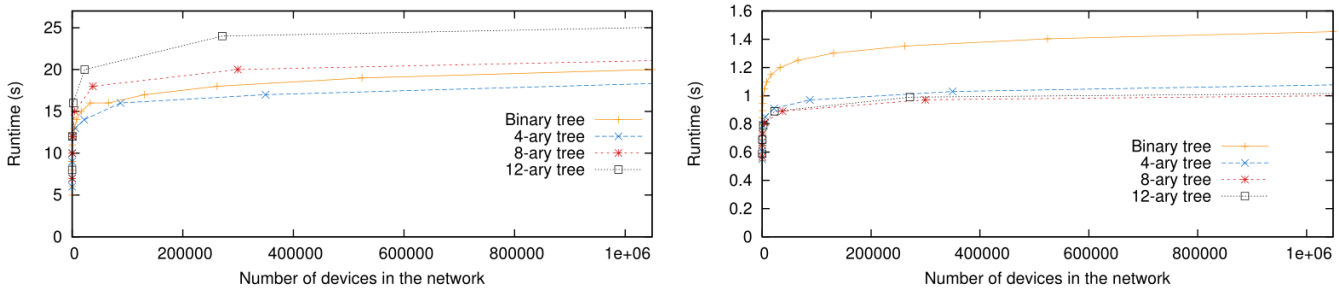
$$E_i \leqslant (56 + 68g_i)E_{send} + (68 + 56g_i)E_{recv} + (3 + 3g_i)E_{mac} + g_i E_{prng}$$

We estimated the energy consumption of attestation based on the energy costs of communication and cryptographic operations reported for two sensor nodes: MICAz and TelosB [47], which belong to the same class of low-end embedded systems that are targeted by our solution.[3] Our energy consumption estimations are presented in Figure 4.6b. The energy consumption on each device D_i increases linearly with the number of neighbors g_i of this device. Therefore, in applications where all devices have the same number of neighbors, our solution distributes energy consumption evenly across all devices in the network $\mathcal{N}$. Due to the use of public key cryptography, energy consumed by the initiator D_1 is more than all other devices. However, if a different initiator device was used for each execution of the attestation protocol, the energy needed for public key cryptography would be amortized among all the devices in $\mathcal{N}$.

Simulation Results. We used the OMNeT++ [104] network simulator to assess the performance of our solutions for large networks of embedded devices. The attestation protocol was implemented on the application layer, where cryptographic operations were emulated with delays corresponding to real measurments of their execution time on SMART [52] and TrustLite [84]. For our simulations, the average end-to-end delay of the communication between devices was set to 20 ms, which corresponds to the average

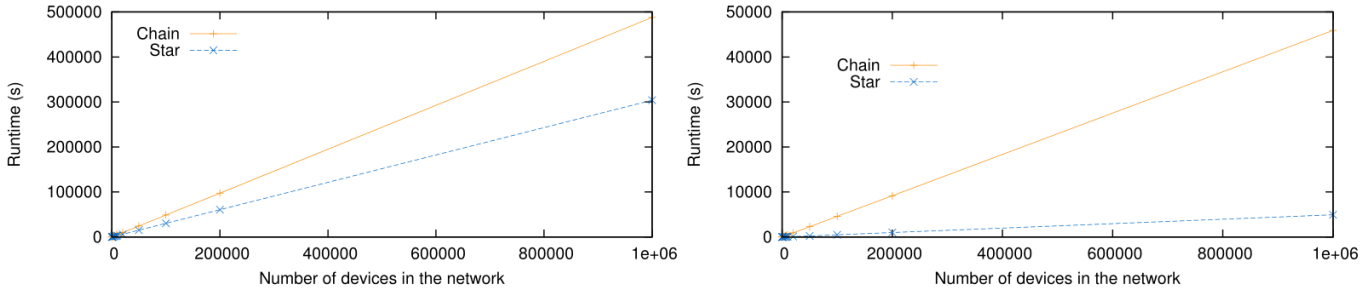
3 SMART and TrustLite are only available as FPGA implementations, which tend to consume more energy than manufactured chips.

delay in sensor networks communicating over ZigBee [135]. Further, we compared our solution to the naïve approach that requires the verifier to attest each device in $\mathcal{N}$ individually. The verification of the attestation response was excluded from the simulation. Note that, while the verification time increases linearly with the size of the network in the naïve approach, it is constant for in our solution. We considered three popular topologies: a star topology, a chain topology, and tree topologies with number of child nodes varying from 2 to 12. Further, we simulated various network sizes, which ranged from 10 to 1,000,000. Figure 4.7, 4.8, and 4.9 show the results of our simulations.



(a) SMART [52] (b) TrustLite [84]

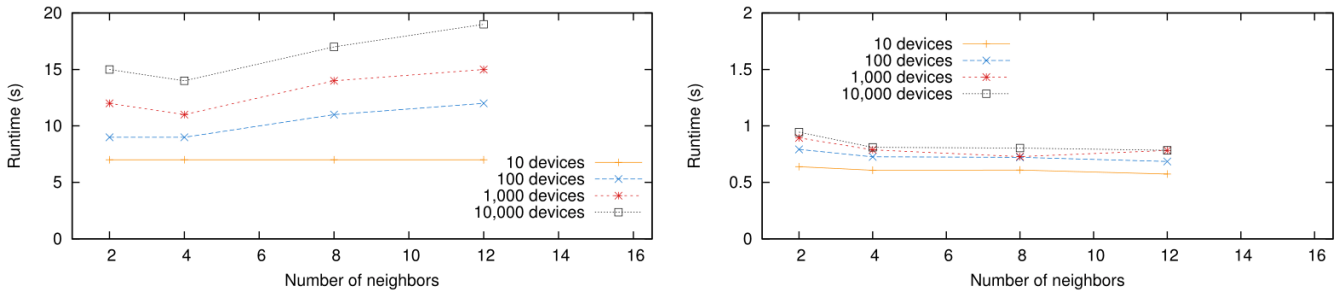
Figure 4.7: Performance of attestation for tree topologies



(a) SMART [52] (b) TrustLite [84]

Figure 4.8: Performance of attestation for chain and star topologies

Runtime of our solution is linear in the size of the network for star and chain topologies (see Figure 4.8), and logarithmic for tree topologies (see Figure 4.7). The runtime of our solution as function of the number of neighbor per device is shown in Figure 4.9 for networks with different sizes. The figure shows how the number of neighbors per device can be optimized with respect to a given network size. For example, in Figure 4.9a) the attestation runtime in networks with 10,000 devices decreases with the number of neighbors, then it starts to increase when it reaches a certain threshold. This happens because the number of neighbors per device affects the performance of attestation in two ways. First, it increases the runtime on individual devices, which have to verify more MACs. This leads to longer overall runtime. Second, it increases the fan-out of the spanning tree, thus decreasing its height. This leads to shorter overall runtime. Therefore, the increase

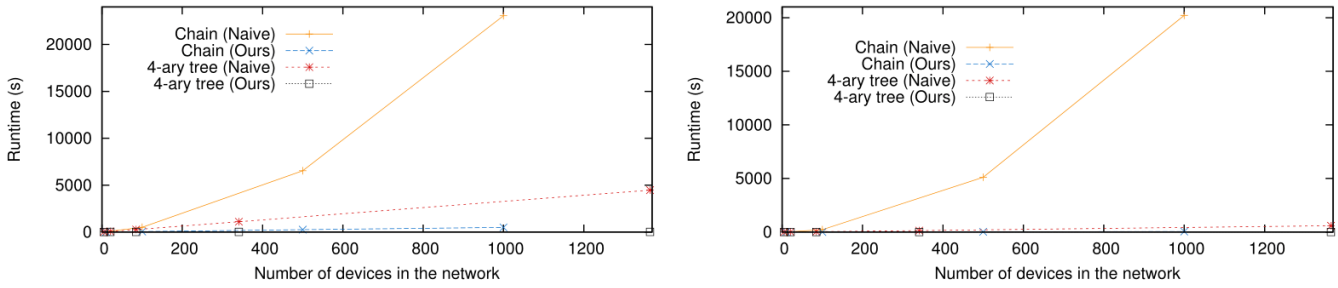


(a) SMART [52] (b) TrustLite [84]

Figure 4.9: Performance of attestation for networks with tree topologies and varying numbers of neighbors per device

in number of neighbors per devices increases the efficiency of attestation up until individual runtime overshadows the benefit of lower height. Note that, this optimal number is dependent on device and network characteristics, i.e., network delays and computational power of devices. The figure shows that for SMART devices, the optimal number of neighbors per device is 2 in networks with 10 to 100 devices, and 4 in networks with 1,000 to 10,000 devices.

A comparison between our solution and the the naïve approach, which requires the verifier to attest each device in $\mathcal{N}$ individually, is shown in Figure 4.10. Our solution performs significantly better than the naïve approach, whose performance is quadratic in the network size for all simulated topologies, i.e., tree, chain, and star topologies.



(a) SMART [52] (b) TrustLite [84]

Figure 4.10: Performance of our attestation compared to the naïve approach

Further, we compare our solution based on interactive and non-interactive attestation. The results are shown in Figure 4.11 for binary trees of up to 1,000,000 devices. Non-interactive attestation shows a significant performance improvement, which can be as high as 28% in very large networks. This performance improvement comes at the cost of additional hardware requirements and assumptions regarding the network topology and the use case scenario.

Our simulation shows that the best performance of our solution is in topologies that allow constructing a spanning tree with a limited fan-out. This is due to the fact that

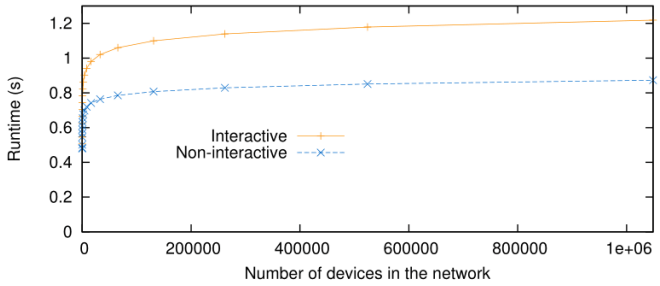


Figure 4.11: Comparison between interactive and non-interactive collective attestation

spanning trees enable parallelism, while the limited fan-out limits the verification overhead on individual devices in the network. In topologies where spanning trees cannot be established, such as chain and star topologies, our attestation is still significantly better than attesting each device individually. Finally, we discuss in Section 4.1.6 an extension of our scheme, which is based on random sampling and is capable of reducing the overhead of attestation for such topologies.

4.1.5 *Security Analysis*

The goal of our attestation solution is to enable the verifier V to check the software integrity of all devices in a network $\mathcal{N}$, i.e., V should accept the attestation report and return $b = 1$ if every device D_i in $\mathcal{N}$ has an unmodified software correctly certified by the network operator O. We formalize this goal as a security experiment $\mathbf{Exp}_{\mathcal{A}}$, where the adversary $\mathcal{A}$ can interact with every device in $\mathcal{N}$ as well as V. $\mathcal{A}$ has full control over the communication channel between every two devices in $\mathcal{N}$, and between $\mathcal{N}$ and V, i.e., it can eavesdrop on, modify, drop, and inject arbitrary messages to any $D_i \in \mathcal{N}$ and to V. $\mathcal{A}$ maliciously modifies of at least one device in $\mathcal{N}$. At the end of the experiment, V outputs the result of attestation b indicating whether it has accepted the attestation report or not, following a polynomial number (in ℓ_N, ℓ_q, ℓ_{mac}, and ℓ_{sign}) of steps performed by $\mathcal{A}$. The output of V represents the result of this security experiment, i.e., $\mathbf{Exp}_{\mathcal{A}} = b$. In the following we provide the definition of secure collective attestation under the predescribed adversary model:

Definition 4.1 (Secure collective attestation under software attacks)**.** *Let f be a polynomial function in ℓ_N, ℓ_q, ℓ_{mac}, and ℓ_{sign}. We consider a collective attestation scheme to be secure under software attacks if the probability* $\Pr\left[b = 1 | \boldsymbol{Exp}_{\mathcal{A}}(1^{\ell}) = b\right]$ *is negligible in* $\ell = f(\ell_N, \ell_q, \ell_{\mathsf{mac}}, \ell_{\mathsf{sign}})$.

Theorem 4.1 (Security of our attestation solution)**.** *The attestation solution presented in this section is a secure collective attestation scheme (Definition 4.1) if the underlying MAC and signature schemes are selective forgery resistant.*

Proof sketch of Theorem 7.1. Let pk_1, c_1, $\mathrm{cert}(pk_1)$, $\mathrm{cert}(c_1)$ be the public key, the reference software configuration, the identity certificate, and the software configuration certificate of the initiator D_1 respectively. As a response to a challenge containing the nonce N_1,

the verifier V receives from D_1 a message $(\beta_1, \tau_1, \sigma_1, \text{cert}(pk_1), \text{cert}(sk_1))$. V accepts and returns $b = 1$ only if $\text{versig}(pk_1; N_1 \| \beta_1 \| \tau_1 \| c_1, \sigma_1) = 1$, and $\beta_1 = \tau_1 = n - 1$. Consider the two cases where $\mathcal{A}$ either maliciously modifies the (1) the software of the initiator D_1, or (2) the software of any other device D_j in the network $\mathcal{N}$. It is easy to see that all possible attacks by $\mathcal{A}$ compromising the software of at least on device in $\mathcal{N}$ is covered by a combination of these cases.

We first consider the case where $\mathcal{A}$ compromises D_1. Due to hardware security architecture, $\mathcal{A}$ is not capable of tampering with the code of attest responsible for measurement and reporting of D_1's software. Consequently, the software configuration c_1' included in $\sigma_1 = \text{sign}(sk_1; N_1 \| q \| \beta_1 \| \tau_1 \| c_1')$ will be different than the reference software configuration c_1 of D_1. In order to convince V to accept and return $b = 1$, $\mathcal{A}$ then must forge the signature σ_1. However, since the signature scheme is selective forgery resistant, the probability of $\mathcal{A}$ forging σ is negligible in ℓ_{sign}.

We now consider the case where $\mathcal{A}$ compromises D_j. Assume that the device D_i is the parent of D_j in the spanning tree, i.e., D_i attest D_j through attdev. Similar to the case of initiator, the secure hardware of D_j prevents $\mathcal{A}$ from tampering with the code of attdev responsible for measurement and reporting of D_j's software. Consequently, the software configuration c_j' included in $\mu_0 = \text{mac}(k_{ij}; N_i \| q \| c_j')$ will be different than the reference software configuration c_j of D_j. In order to make up for the fact that attestation of D_j was not successful, $\mathcal{A}$ then must forge the MAC μ_0. However, since the MAC scheme is selective forgery resistant, the probability of $\mathcal{A}$ forging μ_0 is negligible in ℓ_{mac}. $\mathcal{A}$ could also compensate for this failure in attesting D_j by either decreasing τ_j, increasing β_j, or double counting one of the benign devices D_b. However, since the code of attdev responsible for authenticating β_j and τ_j with $\mu_1 = \text{mac}(k_{ij}; N_i \| q \| \beta_j \| \tau_j)$ is protected by hardware on D_j, $\mathcal{A}$ would have to forge μ_1 which is negligible in ℓ_{mac}. Similarly, to change β_j and τ_j, $\mathcal{A}$ may try to change (β_a, τ_a) that are involved in the computation of β_j and τ_j. However, this is negligible in ℓ_{mac}. Finally, as every attestation report includes a global session identifier q, $\mathcal{A}$ trying to make a device D_b report twice for the same value of q is also negligible in ℓ_{mac}.

Therefore, the probability of $\mathcal{A}$ convincing V to accept the attestation report and return $b = 1$ after maliciously modifying the software of at least one device in $\mathcal{N}$ is negligible in ℓ_{mac} and ℓ_{sign}. □

4.1.6 *Protocol Extensions*

We now decibel different extensions of our solution that enable more services, provide better efficiency, or go beyond the threat model described in Section 4.1.1.

Identification of Compromised Devices. The solution we present assures the verifier V that none of the devices in the network $\mathcal{N}$ has been software compromised. In fact, it reports to V the number of successfully attested devices in $\mathcal{N}$. However, in certain scenarios it might be crucial to pinpoint devices with compromised software. This can be easily done by modifying the code of attdev, to also record the identifier of devices that failed attestation and append it to the attestation report. Consequently, the attes-

tation report accumulated by the initiator D_1 will include of a list of all devices in $\mathcal{N}$, whose software integrity could not be verified. Clearly, this approach increases the message complexity, and hereby the communication overhead, of our attestation solution. It is best suited for scenarios where compromised devices are expected to have a small count. In Section 4.2 we show another collective attestation solution that is capable of identifying compromised devices.

Support for Devices of Different Priority. In the presented solution all devices are assumed to be of the same importance. However, in some scenarios, certain devices might be considered more critical than others. For example, a cluster head in a Wireless Sensor Network (WSN) is much more important than a regular sensor node, and its software integrity is hereby more critical than that of a sensor node. By weighting the attestation results of devices, our solution enables supporting such scenarios. In particular, when a critical device is attested successfully by a device D_i, the counters β_i and τ_i of D_i are incremented by a weighted factor that reflects the importance of that attested device. For example, a device that is at least three times as important as any regular device in $\mathcal{N}$ will lead to an increase of 3 to β_i and τ_i when attested successfully.

Random Sampling. One way to enhance the performance of our solutions is to use random sampling, i.e., verify the integrity of a small subset $\mathcal{N}' \subset \mathcal{N}$, which is statistically representative of $\mathcal{N}$. This is particularly useful for worst case scenarios, such as chain and star topologies. Random sampling provides V with probabilistic assurance about the integrity of a certain number of devices in $\mathcal{N}$, i.e., V learns that with probability p at least x devices in $\mathcal{N}$ are running a benign software certified by the network operator O. This extension executes as follows: Along with the nonce N, V sends in the challenge of attest the desired size z of the sample $\mathcal{N}'$. This size z is broadcasted across the network along with q through attdev. Every device $D_j \in \mathcal{N} \setminus \{D_1\}$ uses a global deterministic function f and the parameters z and n to determine whether it is part of the sample $\mathcal{N}'$. Devices also use f to find out whether their children in the spanning tree need to be attested. Finally, V only receives the attestation results of devices in $\mathcal{N}'$. This approach provides V with assurance that the attestation results of $\mathcal{N}'$ reflect, with certain confidence interval and confidence level, the actual state of $\mathcal{N}$. Consider a network $\mathcal{N}$ with 10^5 devices. By attesting a sample $\mathcal{N}'$ including 9% of $\mathcal{N}$'s devices, V obtains a confidence interval of 1% and confidence level of 95%.

Software Updates. Our solutions can also be used to check correctness of software updates. In particular, each software update includes a new software configuration certificate $\text{cert}(c_{\text{new}})$. This certificate is authenticated by the updated devices D_i based on the keys in $\mathcal{K}_i$ and shared with all of its neighbors. If certificate verifies successfully, neighbors replace the old reference software configuration of D_i with c_{new}. Finally, to ensure that the software of D_i was updated successfully, D_i can be either attested by a remote verifier V based on its secret key sk_i, or by one (or more) neighbors using key(s) from $\mathcal{K}_i$. Further, while attesting the network, V can detect roll-back attacks, where $\mathcal{A}$ installs old (probably vulnerable) software versions.

Highly dynamic swarms. Our solution assumes that the topology of $\mathcal{N}$ is static during the execution of the attestation protocol. However, it may leverage an existing routing

protocol to allow attestation of highly dynamic networks, where the topology changes are often and may occur during attestation execution. The main idea is to generate a a *virtual* spanning and count on the routing protocol to ensure delivery of protocol messages from child to parent nodes. This approach leads to a higher communication overhead as messages between neighboring devices are instead sent through multiple hops.

Mitigation of Denial of Service (DoS) Attacks. The design of our solutions, which is mostly based on symmetric cryptography, makes it a less attractive target for DoS attacks. However, the protocols that are based on asymmetric cryptography, i.e., join and attest, can be still targeted by DoS attack. For instance, $\mathcal{A}$ can make a compromised device repeatedly send fake certificates to its neighbors as part of join. This will exhaust the resources of the neighbors that have to perform computationally expensive asymmetric cryptography to verify the certificates. We imagine two solutions for thwarting DoS on join. The first solutions is to limit the frequency of execution of join, while the second involves executing join with low priority. Current lightweight security architecture for embedded networks provide support for real-time execution, e.g., TyTAN [28]. Consequently, some events, such as join execution can be handled with lower priority, allowing the resources of the device to be allocated to other tasks. This ensures that the CPU dedicates only otherwise idle cycles for executing join protocol. DoS attacks on attest are also critical as $\mathcal{A}$ can execute a global DoS on the whole network through one device, i.e., by sending one attestation request. In Section 4.2 we show how such attacks can be thwarted.

Physical Attacks. In some applications, such as IoT networks we consider in this section, it might be reasonable to assume no physical attacks on any of the devices in the network. However, other applications involve devices that can be the target of physical attacks. Our solution assumes that none of the devices can be physical attacks. An adversary $\mathcal{A}$ who is capable of physically attacking one device can undermine the security of our solution and evade detection of an arbitrary number of compromised devices. Several mitigation techniques may serve for thwarting $\mathcal{A}$. One can use Physical Unclonable functions (PUFs) as the primitive for authenticating attestation reports. PUFs are considered tamper-resistant and can aid in mitigating physical attacks. Another approach is absence detection to detect all physically attacked devices. We elaborate on this approach in Chapter 5, where we describe our collective attestation solution that detect physical attacks. Furthermore, the collective attestation solution we describe in Section 4.2 is more resilient to physical attacks, since a physically attacked device can only evade its own detection by that solution.

4.1.7 *Conclusion*

The solution we presented in this section represents the first attestation solution that allows efficient and scalable attestation of networks formed of a very large number of heterogeneous embedded devices. This solution represents the most efficient multi-device attestation as it allows attesting a million-device network in order of seconds. The

security of this solution is analyzed in the first security model for collective attestation, which adheres to the security model of single-device attestation. Note that, this solution is most suitable for networks, where none of the devices can be physically attacked. It considers DoS attacks on the attestation protocol to be out of scope. And, it allows the verifier to learn only the number of devices in the network that were attested successfully. Based on these properties that adhere to single-device attestation, this solutions is able to guarantee best efficiency. In Section 4.2 we present another solution that allows detection malware infestation in large networks. However, the solution we present in Section 4.2 is resilient to DoS and physical attacks, and is capable of identifying compromised devices while imposing minimal additional overhead.

4.2 SECURE AND SCALABLE AGGREGATE NETWORK ATTESTATION

The solution we presented in Section 4.1 constitutes a first step towards secure and scalable attestation for large embedded networks. However, to achieve best efficiency, this solution adheres to the adversary model of single-device attestation, i.e., it assumes a software-only adversary that is not capable of performing physical attacks. This assumption allows using hop-by-hop authentication and attestation based in symmetric key cryptography. Further, efficiency is also enhanced by only collecting the number of devices that were not attested successfully and not their identity or software configuration. While the design of this solution increases its efficiency considerably and hereby its scalability, it imposes several assumptions that limits its applicability. For example, hop-by-hop attestation requires trusting all intermediaries, i.e., every device involved in the protocol should participate in the attestation process, and should be equipped with a lightweight security architecture as described in Section 4.1.1. Moreover, an adversary that is capable of tampering with the lightweight security architecture of a small number of devices can undermine the security of the proposed solution. All these assumptions and requirements limits the applicability of this solutions to scenarios where all devices are trusted (i.e., owned) by the same entity, e.g., smart home/office.

In this section we present a second collective attestation solution for centrally managed systems of heterogeneous devices. However, while this solution imposes additional overhead, it imposes less assumptions, provides stronger security guarantees, and is applicable to a wider range of applications, e.g., scenarios where devices involved in the attestation protocol are not all deployed/trusted by the same entity. The security of the scheme is based on a novel signature scheme that allows: (1) scalability, i.e., can be efficiently applied to large networks of embedded devices; (2) public verifiability, i.e., can be verified by any entity that holds the public key; and (3) untrusted intermediaries, i.e., devices that are not involved in signing do not have to be trusted. As mentioned earlier, the two solutions presented in this chapter complete each other. While Section 4.1 provides best efficiency, this section provides applicability to a wider range of application with strong security guarantees.

Contribution. We investigate security of a large embedded networks that have a stronger adversary model and different requirements. Then we devise a secure collective attesta-

tion solution which satisfies the requirements in these networks. Our solution is based on a novel signature scheme that we design, which combines aggregate and multisignatures providing both scalability and heterogeneity.[4] In particular, the devised signature scheme allows aggregating signatures on attestation reports of heterogeneous devices, with an overhead on the verifier that is constant in the number of devices in the network. We denote this signature scheme by Optimistic Aggregate Signature (OAS). The attestation solution leverages OAS to (1) enable applicability to large scale IoT deployments, where not all devices can be trusted, e.g., cloud servers and router; and (2) provide resiliency against physical attacks. In order to demonstrate feasibility, we also show how to instantiate this solution on the two recent security architecture for low-end embedded devices used in Section 4.1, i.e., SMART [52] and TrustLite [84]. Moreover, we present extensive performance evaluation based on these two instantiations, in addition to simulations of the solution in networks of up to $1,000,000$ devices demonstrating scalability. Assuming the devices that perform the aggregation to be untrusted, this solution allows attestation a million-device network in 2.5 seconds.

Outline. We provide a brief overview of our solution in Section 4.2.1, present our OAS signature scheme in Section 4.2.2, and provide details of our solution in Section 4.2.3. Our implementation on SMART and TrustLite is described in Section 4.2.4. In Section 4.2.5 we present our performance evaluation, and we examine security of this solution in Section 4.2.6. An extension to the solution is described in Section 4.2.7, and this section concludes in Section 4.2.8.

4.2.1 *Collective Attestation*

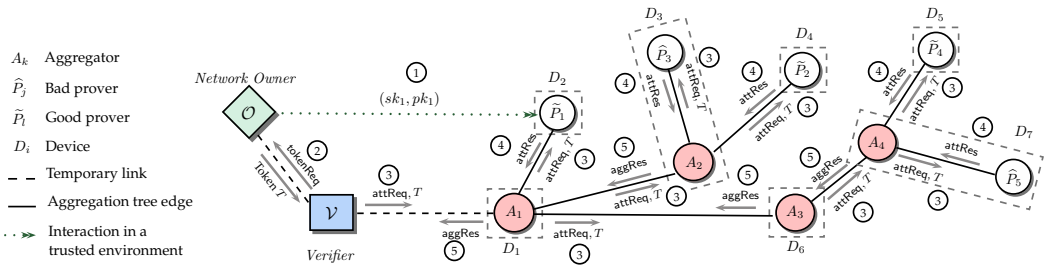


Figure 4.12: Example 7-device network (four aggregators and five provers)

4.2.1.1 *Problem Description and System Model*

As in Section 4.1.1, we consider a centrally managed dynamic network $\mathcal{N}$ that is formed of a large number n of heterogeneous devices. O is the owner/operator of $\mathcal{N}$, and V is an entity that is interested in assessing the trustworthiness of $\mathcal{N}$. While a routing protocol is not assumed, the devices in $\mathcal{N}$ should at least be capable of communicating to their neighboring devices [44, 68, 114, 115]. Devices' mobility might be involuntary. Therefore,

4 Combining aggregate and multisignature and devising our own signature scheme was necessary as existing schemes did not satisfy the requirements of collective attestation.

neither operator O nor the verifier V are assumed to be aware of the network topology at any given time.

An example network is shown in Figure 4.12. We consider four logical entities participating in the attestation protocol: *operator* (O), *verifier* (V), *aggregator* (A), and *prover* (P). A prover P_i is related to a specific device, and can be seen, for example, as a software component running on it. The role of a prover is to compose the attestation response, i.e, an integrity proof of the software of that device, which is then sent through a set of aggregators to V. Provers are heterogeneous, i.e., they may have different hardware and software. However, it is assumed that most of the provers have the most recent version of a benign software. These provers are denoted by *good provers* $\widetilde{P}_i$. The remaining provers that have outdated or compromised software are referred to as *bad provers* $\widehat{P}_i$. Similar to a prover, an aggregator A_i can be seen as a software components running on a device. The purpose of an aggregator, however, is only to collect and aggregate attestation responses generated by other entities, i.e., other provers or aggregators. O is the entity responsible for initialization, deployment, and maintenance of every prover P_i in $\mathcal{N}$. O is not responsible for aggregators, which could be owned/deployed by other entities. A physical device can be formed of one or more of these logical entities, e.g., D_7 in Figure 4.12 is formed of both the aggregator A_4 and the prover $\widehat{P}_5$. Device D_1, on the other hand, is only formed of the aggregator A_1.

In this model, a collective attestation solution enables a verifier V (which can be remote or local) to assess the trustworthiness of $\mathcal{N}$ in a secure, scalable, and efficient manner. The basic idea is that, if none of the provers has been physically attacked, we consider $\mathcal{N}$ to be trustworthy if all provers have the most recent version of a benign software that is accepted by V. Further, the collective attestation solution should allow V to pinpoint compromised devices. Finally, although physically attacked devices can evade detection, which is also the case when attesting each device individually, a physically attacked devices should not be capable of helping other devices evade detection by collective attestation.

4.2.1.2 *Requirements Analysis*

Objectives. Based on the model described above, a collective attestation solution for large networks of embedded devices should provide, in addition to the five properties described in Section 4.1.1, the following properties:

- *Property #1:* Guarantee for every prover whose hardware has not been maliciously tampered with, that the attestation report generated by this prover reflects the actual software state of the prover during the execution of the attestation protocol. We refer to this property as the *Unforgeability* property.

- *Property #2:* Ensure for every prover whose hardware has not been maliciously tampered with, that the attestation report generated by this prover was incorporated into the aggregate attestation report received by the verifier. This property is denoted by the *Completeness* property.

- *Property #3:* Allow the verification of the aggregated attestation report sent to the verifier by any other entity. We refer to this property as *Public Verifiability*. Note that, for public verifiability, the verification of the attestation report ensures that software trustworthiness of $\mathcal{N}$ at a given point in time between creation of the attestation challenge and the receipt of the attestation report from V.
- *Property #4:* Ensure applicability to networks where devices are heterogeneous, and enable the usage of every integrity measurement adopted by the devices in $\mathcal{N}$. We refer to this property as *Heterogeneity*.
- *Property #5:* Guarantee the generation of an aggregate attestation report when all the devices in $\mathcal{N}$ are available and all the communication links are up. This property is denoted by *Availability*.
- *Property #6:* Provide no additional Denial of Service (DoS) attack vector for the adversary, i.e., the adversary should not be capable of running a global DoS attack on the network via a single device.

The satisfaction of property #1 and property #2 constitute the main security objective of collective attestation. Moreover, a secure collective attestation solution should also satisfy property #6. Property #3 and property #4 are needed when the configuration of the network should be hidden from the verifier. It is achieved through digital signatures. Finally, property #5 is required for supporting new device types and is achieved through combining aggregate and multisignatures in our novel Optimistic Aggregate Signature (OAS) scheme.

Adversary Model. As in Section 4.1 we assume $\mathcal{A}$ can eavesdrop on, modify, drop, or replay any message exchanged between all devices in $\mathcal{N}$ and between any device D_i and the verifier V. We assume that both O and V are trusted. However, we consider two kinds of adversaries: The software-only adversary from single-device attestation, which can maliciously modify all the software on any device D_i in $\mathcal{N}$ except what is protected by hardware. And, a physical attacker that can tamper with the hardware of any aggregator entity A_i, i.e., the attacker can tamper with the hardware of any device acting as aggregtor, and modify the software or extract the secrets of the aggregator logical entity that are protected by hardware. Finally, we assume a stealthy adversary that aims at evading detection. Hence we consider DoS attacks to be out of scope. Nevertheless, we limit such attacks by not allowing $\mathcal{A}$ to run a global DoS on $\mathcal{N}$ through one device D_1.

Device Requirements. The design of our solution aims as satisfying all properties #1 to #6 as well as properties #1 to #5 described in Section 4.1.1. However, in order to satisfy these properties, it should be possible to remotely attest each prover in $\mathcal{N}$, i.e., every device D_i acting as a prover should satisfy the requirement for secure remote attestation as discussed in Section 4.1.1. A prover that does not satisfy these requirements can evade detection by our attestation solution.

Assumptions. Devices can have different hardware and/or software. However we assume that every prover (i.e., every device D_i that acts as prover) in $\mathcal{N}$ satisfies the requirements for secure remote attestation. We assume that all devices can communicate,

i.e., devices can at least communicate to their direct neighbors. Further, throughout the execution of the attestation, the network is assumed to be connected and its topology has to remain static. More importantly, each device D_i in $\mathcal{N}$ should be reachable from any other device D_j. Finally, we assume that all cryptographic primitives, such as the Optimistic Aggregate Signature (OAS), and their implementations are secure.

Protocol Overview. The concept of our solution is illustrated in Figure 4.1 in a setting where $\mathcal{N}$ is formed of seven physical devices (D_1 through D_7). The setting incorporates the following logical entities: the operator O, the verifier V, the five provers ($\widetilde{P}_1$ through $\widehat{P}_5$), and the four aggregators (A_1 through A_4). Our solution consists of three protocols: device initialization, token request, and attestation. O initializes every device (e.g., D_2 in Figure 4.12) with the cryptographic keys required for executing the attestation protocol. This initialization is denoted by init (operation ① in Figure 4.12). init is performed by O in a secure environment.

Only a verifier V that has an valid attestation token can attest $\mathcal{N}$ at any given time. The attestation token is generated by O, and is securely transferred to V through the tokenReq protocol (operation ②). For verifying $\mathcal{N}$, V picks a random device as an initiator D_ι, which acts as the root aggregator A_1. V then sends the attestation challenge, which includes the attestation token, to A_1 (operation ③). This challenge is flooded across $\mathcal{N}$, and a spanning tree is formed rooted at A_1. The tree constitutes of aggregtors as intermediate nodes, and provers as leafs. Note that, an edge in the spanning tree may represent a communication link between two devices, or a virtual link between an aggregator and a prover residing on the same device (e.g., between $\widetilde{P}_3$ and A_2).

Eventually, the provers ($\widetilde{P}_1$, $\widehat{P}_2$, $\widetilde{P}_3$, $\widehat{P}_4$ and $\widetilde{P}_5$ in the figure) generate a measurement of their software configuration and send it the their parents as their attestation response (operation ④). These responses are then collected and aggregated by the aggregators (A_1, A_2, A_3 and A_4), which forward the aggregation results to their parents and so on (operation ④). A final aggregated report is then sent by A_1 to V (operation ⑤), which then verifies it and learns the identities of all compromised devices. This protocol is referred to as attest.

4.2.2 *Proposed Signature Scheme*

Before going into the details of our attestation solution we first present *Optimistic Aggregate Signatures* (OAS). OAS is a generalization of multisignatures and aggregate signatures that allows signing multiple messages $m_1, \ldots, m_n$ by n different signers. Messages $m_1, \ldots, m_n$ can be different. However, most of them are expected to be equal to the "default" message M. OAS allows aggregating multiple signatures into a single aggregate signature. In the optimisitic case where most of the signed messages are equal to M, the generated aggregate signature is considerably shorter, and has less verification time than the separate n signatures. Concretely, the verification time and signature size in an OAS scheme should be constant in the number of signatures on the default message M. In Section 4.2.2.2 we present an instantiation of OAS, where the size of aggregate signatures increases linearly with the number of messages that are not equal to the de-

fault message M and with the number of different signers that signed these messages. The time required to verify an aggregate signature is constant increases linearly with the number of messages different from M, while is constant in the number of signers of these messages.

4.2.2.1 *Definition of an OAS*

We now formally define OAS (Definition 4.2), and define its correctness (Definition 4.3) and unforgeability property (Definition 4.4). Note that, OAS is considered a basic block for secure collective attestation as it satisfies both heterogeneity and scalability properties.

Definition 4.2 (Optimistic Aggregate Signatures)**.** *We define an* Optimistic Aggregate Signature (OAS) scheme *as a tuple of probabilistic polynomial time algorithms denoted by* $(\mathsf{genkeysign}, \mathsf{aggPK}, \mathsf{sign}, \mathsf{aggsign}, \mathsf{veraggsign})$. *Algorithm* genkeysign *takes a security parameter* $\ell_{\mathsf{sign}} \in \mathbb{N}$ *as input, and outputs for each device* D_i *a key pair* (sk_i, pk_i)*, where* sk_i *is the secret signing key and* pk_i *is the public verification key, i.e.,* $(sk_i, pk_i) \leftarrow \mathsf{genkeysign}(1^{\ell_{\mathsf{sign}}})$. *The security parameter* ℓ_{sign} *determines the message space* M *of the signature scheme. Algorithm* aggPK *takes a set* $\{pk_1, \ldots, pk_n\}$ *of public keys as inputs, and outputs the aggregate public key apk, i.e., apk* $\leftarrow \mathsf{aggPK}(pk_1, \ldots, pk_n)$. *Algorithm* sign *takes a message* $m \in M$*, the default message* $M \in \{0,1\}^*$*, and secret key* sk_i *of* D_i *as input, and outputs a signature* α_i *on* m*, i.e.,* $\alpha_i \leftarrow \mathsf{sign}(sk; m, M)$. *An individual signature* α_i*, generated by* sign *over message* m *using* sk_i*, is considered in OAS as an aggregate signature that contains a single signature over* m. *Algorithm* aggsign *takes as input two OAS signatures* α_i, α_j *and the default message M, and outputs a new OAS signature* α_k*, which contains all the signatures in* α_1 *and* α_2*, i.e.,* $\alpha_k \leftarrow \mathsf{aggsign}(\alpha_i, \alpha_j, M)$. *Finally, on input of an OAS signature* α_i*, an aggregate public key apk, the default message M, and the set of public keys* $S_\perp$ *that are in apk but did not contribute to the generation of* α_i*, algorithm* veraggsign *outputs* $\perp$ *indicating that the verification failed, or the set* $\mathcal{B} = \{(m_i, S_i) : i = 1, \ldots, \omega\}$*, which shows for each message* m_i *all the public keys* $pk_i \in S_i$ *that signed this message otherwise.*

An intuitive definition of the correctness of an OAS scheme can be described as follows: when all the signers that are involved in the scheme are honest contributing at most one correct signature to the aggregate signature, the verification will output the correct set, which associate each signer to the message it signed. Because of the many possible orders in which one can aggregate signatures, it becomes increasingly tedious to provide a formal definition of the correctness of OAS. For simplification, we provide the following notation. Let $\mathcal{B}_i, \mathcal{B}_j$ be two sets that contain the tuples $(m, S) \in \{0,1\}^* \times (\{0,1\}^*)^*$, we denote by $\mathcal{B}_k = \mathcal{B}_i \sqcup \mathcal{B}_j$ the "merged" set of tuples (m, S), such that $S = S_i \cup S_j$ if $\exists (m, S_i) \in \mathcal{B}_i$ and $\exists (m, S_j) \in \mathcal{B}_j$, where $S = S_i$ if $\exists (m, S_i) \in \mathcal{B}_i$ and $\nexists (m, S_j) \in \mathcal{B}_j$, and where $S = S_j$ if $\exists (m, S_j) \in \mathcal{B}_j$ and $\nexists (m, S_i) \in \mathcal{B}_i$.

Definition 4.3 (Correctness of OAS)**.** *An Optimistic Aggregate Signature (OAS) scheme is correct if:*

- *The signing works correctly, if* $\forall m, M \in \{0,1\}^*, \forall \ell_{\mathsf{sign}} \in \mathbb{N}, \forall (pk_i, sk_i) \leftarrow \mathsf{genkeysign}(\ell_{\mathsf{sign}})$*, and* $\forall S_\perp / pk \notin S_\perp$ *the following holds: if* $m = M$*,* $\mathsf{veraggsign}(apk, S_\perp, \alpha, M)$ *outputs* $\emptyset$*;*

and if $m \neq M$, $\mathsf{veraggsign}(apk, S_\perp, \alpha, M)$ *outputs* $\{m, \{pk\}\}$ *whenever* $\alpha \leftarrow \mathsf{sign}(sk, m, M)$ *and* $apk \leftarrow \mathsf{aggPK}(S_\perp \cup \{pk\})$.

- *The aggregation works correctly, i.e., for all disjoint sets* S_1, S_2, *all OAS signatures* α_i, α_j, *all default messages* $M \in \{0,1\}^*$, *and all subsets* $S_{\perp,1} \subseteq S_1$ *and* $S_{\perp,2} \subseteq S_2$, *it should hold that* $\mathsf{veraggsign}(apk, S_{\perp,i} \cup S_{\perp,j}, \alpha_k, M) = \mathcal{B}_i \sqcup \mathcal{B}_j$ *if* $\mathsf{veraggsign}(apk_i, S_{\perp,i}, \alpha_i, M) = \mathcal{B}_i$ *and* $\mathsf{veraggsign}(apk_j, S_{\perp,j}, \alpha_j, M) = \mathcal{B}_j$ *for* $apk_i \leftarrow \mathsf{aggPK}(S_i)$, $apk_j \leftarrow \mathsf{aggPK}(S_j)$, *and* $apk \leftarrow \mathsf{aggPK}(S_i \cup S_j)$.

Definition 4.4 (Unforgeability of OAS)**.** *Unforgeability of OAS implies that the adversary is not capable of generating an OAS signature* α *that associates to an honest signer a message that it has never signed, even when all the other involved signers are dishonest. Formally, the experiment below should with negligible probability return 0 for every polynomial time adversary* $\mathcal{A}$*:*

$(pk, sk) \leftarrow \mathsf{genkeysign}(\ell_{\mathsf{sign}})$
$(\alpha, S_\perp, (pk_1, \ldots, pk_n), (sk_1, \ldots, sk_n)) \leftarrow \mathcal{A}^{\mathsf{sign}(sk,\cdot)}(pk)$
If $\exists\, i : pk_i \neq pk \wedge (pk_i, sk_i) \notin \mathsf{genkeysign}(\ell_{\mathsf{sign}})$ *then return 0*
Let $S \leftarrow \{pk_1, \ldots, pk_n\}$
$apk \leftarrow \mathsf{aggPK}(S)$
$\mathcal{B} \leftarrow \mathsf{veraggsign}(apk, S_\perp, \alpha, M)$
If $S_\perp \not\subseteq S$ *or* $\exists (m_i, S_i) \in \mathcal{B} : S_i \not\subseteq S$ *then return 0*
If $\exists (m_i, S_i) \in \mathcal{B} : pk \in S_i$ *and* $m_i \notin Q$ *then return 1*
Let $S_M \leftarrow S \setminus \left(S_\perp \cup \bigcup_{(m_i, S_i) \in \mathcal{B}} S_i\right)$
If $pk \in S_M$ *and* $M \notin Q$ *then return 1*
Else return 0

where Q *denotes the set of all messages queried by* $\mathcal{A}$ *from the* $\mathsf{sign}(sk, \cdot)$ *oracle.*

For the unforgeability notion in Definition 4.4 it is required that $\mathcal{A}$ knows the keys of every dishonest signer. We model this requirement in the security experiment by requiring $\mathcal{A}$ to return those keys along with his forged signature. Realizing such a requirement can be done in practice by relaying on a trusted third party that generates all the keys, e.g., O, or making every signer execute an extractable proof of knowledge for its secret key. The proof of knowledge can be executed interactively with a trusted third party, or included within the public key. Finally, it has been shown by Ristenpart et al. [113] that some schemes require only minor modifications to allow generating a simple proof of possession of the secret key. This proof is basically a signature on a challenge. These schemes include the Boldyreva's multisignature scheme [24], which we base our OAS construction in Section 4.2.2.2 on. Therefore, the technique described by Ristenpart et al. is also applicable to our OAS construction. Note that, we require in the unforgeability definition that the public key sets $S_\perp$ and S_i should be subsets of $S = \{pk_1, \ldots, pk_n\}$. This check can be performed by the entity that verifies the signature, either by searching for the keys in S, or making each signer provide a proof that *apk* indeed includes its key. This can be done through a certificate for example.

4.2.2.2 *Instantiation from pairings*

We now present an instantiation from pairings for our OAS scheme. This intantiation is a combination of the aggregate signature scheme from Boneh et al. [25], and the multisignature scheme from Boldyreva [24]. Recall that, an aggregate signature scheme, such as Boneh et al.'s scheme, allows signers to sign different messages while having a verification time of an aggregate signature, which is linear in the number of signatures included in it. On the contrary, in a multisignature scheme, such as Boldyreva's scheme, signatures on the same message can only be aggregated. However, the verification time of an aggregate signature is constant in the number of aggregated messages. The construction we present here basically uses Boldyreva's scheme for aggregating signatures on the same message while on top of it leveraging Boneh et al.'s scheme to aggregate the resulting multisignatures. It is easy to see that the algebra easily adds up. However, such a combination should be carefully considered in terms of security. In particular, aggregate signatures are popular for having sophisticated constraint concerning the signed messages, the composition of signers, and key setup. These restrictions may break the security of the signature scheme if not adhered to correctly [22]. A formal proof of unforgeability of our OAS construction according to Definition 4.4 can be found in [12].

Consider two multiplicative groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ with prime order p, where g_1, g_2, g_t are generators $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ respectively. Let $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ be a bilinear map such that e is efficiently computable, $e(g_1^x, g_2^y) = g_t^{xy} \forall x, y \in \mathbb{Z}_p$, and let $\psi : \mathbb{G}_2 \to \mathbb{G}_1$ be an efficiently computable isomorphism such that $\psi(g_2) = g_1$. We denote by $H : \{0,1\}^* \to \mathbb{G}_1$ a hash function mapping bit strings of arbitrary limited length to elements of $\mathbb{G}_1$. H is modeled as a random oracle [23]. Our OAS instantiation is defined as follows:

KEY GENERATION. Each participant chooses a random value $x \in_R \mathbb{Z}_p$ as its secret key *sk*. The public key is then $pk \leftarrow g_2^x$.

PUBLIC KEY AGGREGATION. Individual public keys $pk_1, \ldots, pk_n$ can be aggregated into an aggregate public key $apk = \prod_{i=1}^{n} pk_i$.

SIGNING. A message m is signed as $\alpha \leftarrow (H(m)^x, \emptyset)$ if $m = M$, and as $\alpha \leftarrow (H(m)^x, \{(m, \{pk\})\})$ if $m \neq M$

SIGNATURE AGGREGATION. To aggregate two OAS signatures $\alpha_i = (\tau_i, \mathcal{B}_i)$ and $\alpha_j = (\tau_j, \mathcal{B}_j)$, one should compute $\tau_k \leftarrow \tau_i \cdot \tau_j$, and "merging" the sets $\mathcal{B}_i$ and $\mathcal{B}_j$ into $\mathcal{B}_k$, i.e., $\mathcal{B}_k \leftarrow \mathcal{B}_i \sqcup \mathcal{B}_j$. The output of signature aggregation is the signature $\alpha_k = (\tau_k, \mathcal{B}_k)$.

VERIFICATION. Let M be the default message, *apk* be the aggregate public key, and $S_\perp$ be the set of public keys in *apk* that did not contribute to the generation of an OAS signature $\alpha = (\tau, \mathcal{B} = \{(m_1, S_1), \ldots, (m_v, S_v)\})$. To verify α, compute:

$$apk_M \leftarrow \frac{apk}{\prod_{pk \in S_\perp} pk \cdot \prod_{i=1}^{v} \prod_{pk \in S_i} pk} .$$

then verify that:

$$e(\tau, g_2) \;=\; e\big(H(M), apk_M\big) \cdot \prod_{i=1}^{v} e\big(H(m_i), \prod_{pk \in S_i} pk\big) \;.$$

Return $\mathcal{B}$ if verification succeeded and $\perp$ otherwise.

We emphasize that in order to guarantee unforgeability of this construction, the signers should provide a proof that they possess their secret key. Another option is to rely on a trusted third party for the generation and distribution of these keys. This requirement is shared with other existing multisignature schemes, e.g., [24, 91]. Proof of possession can be achieved in this scheme by signing an arbitrary message, however, based on a different hash function [113],[5] and appending this signature to the public key. It can also be achieved by including a Schnorr signature, that secret keys can be extracted from through applying the generalized forking lemma [19].

4.2.3 *Protocol Description*

Our solution allows the verifier V to distribute an attestation challenge on every prover P_i in the network $\mathcal{N}$. Each prover is then expected to produce a measurement of its software configuration and sign it using our OAS scheme. Signed measurements are aggregated by aggregators A_i en route to V. The solution also involves a verifier authorization scheme that forbid unauthorized verifiers from attesting $\mathcal{N}$. This is required to thwart Denial of Service (DoS) attacks on $\mathcal{N}$.

The solution constitute multiple protocols that are executed between the operator O, V, the provers P_i, and the aggregators A_i in $\mathcal{N}$. These protocols are described in details in the following:

Device Initialization. Before deployment, the network operator O initializes each device D_i in $\mathcal{N}$ with O's public key pk_O, an OAS key pair ($sk_i \in_R \mathbb{Z}_p, pk_i \leftarrow g_2^{sk_i}$), and an identity certificate $\mathsf{cert}(pk_i)$ certifying that D_i with identity id_i has a valid OAS key pk_i to which it securely stores a secret key sk_i. This protocol is denoted by init and is formally:

$$\mathsf{init}(1^\ell) \rightarrow \big(sk_i, pk_i, \mathsf{cert}(pk_i)\big) \;.$$

Token request. Attesting $\mathcal{N}$ requires authorization of the verifier. This is important to mitigate global DoS attacks that can be launched on the entire network through a single device. For this reason, every verifier V that wishes to attest $\mathcal{N}$ should possess a valid attestation token T. The token should be issued by the $\mathcal{N}$'s operator O, and is acquired offline by V through $\mathsf{tokenReq}$ protocol. The main goal of $\mathsf{tokenReq}$ is allowing public verfiability while thwarting DoS attacks.

The main parameter of $\mathsf{tokenReq}$ is the list of monotonic counters $ctr_1, \ldots, ctr_s$ with values $val_1, \ldots, val_s$ that are stored by O. These counters mainly serve for preventing

5 One can also use the same hash if the message space for regular signature is different than that of proofs of possession.

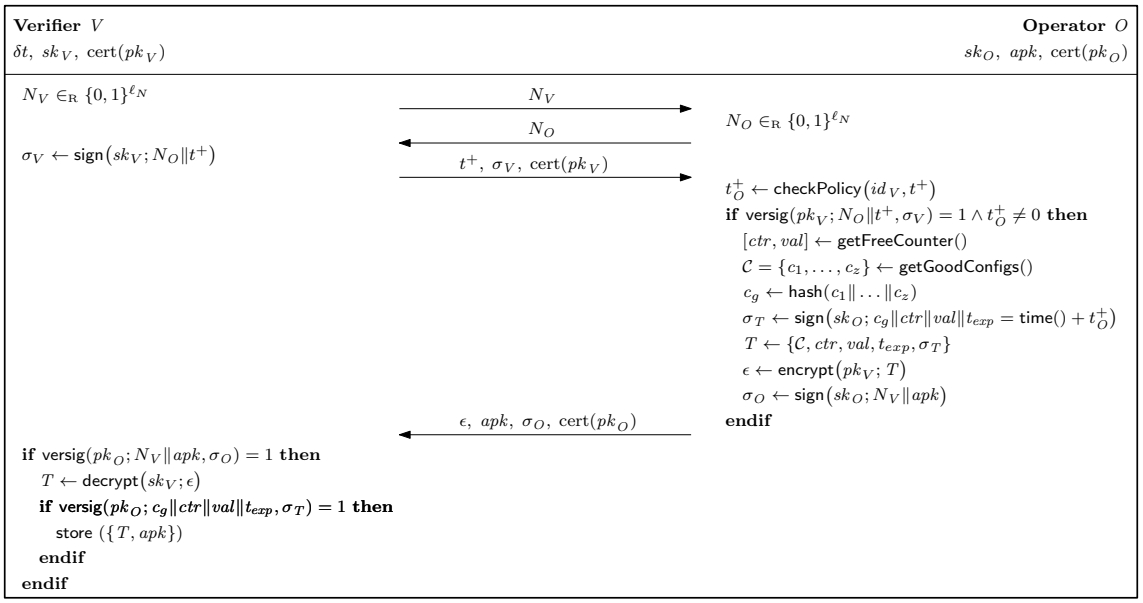


Figure 4.13: Protocol tokenReq

replay attacks on tokenReq. In particular, each counter is assigned for a single token T for a limited amount of time t_{exp} during which this counter is marked as "busy". t_{exp} represents the expiry time of T, and is chosen by V through tokenReq. When tokenReq is initialized by V, O checks for a counter *ctr* that is not busy. The value *val* of *ctr* is then incremented and the tuple (ctr, val) is included in the token. This procedure is denoted by getFreeCounter(). A separate copy of these counters is also kept by each device in $\mathcal{N}$. When an attestation request containing T is received, each device checks the value of the counter in T, before proceeding with the attestation or forwarding the attestation request to neighbors. Attestation is only proceeded if the received value of the counter is greater than the locally stored one. In this case, the local value is replaced with the received one. This procedure is referred to as checkCounter().

The details of tokenReq are shown in Figure 4.13. tokenReq is initiated by V by sending a fresh nonce N_V to O. This nonce reflects V's interest in a token for the attestation of $\mathcal{N}$. O replies with another fresh nonce N_O. As a response to N_O, V generates a signature σ over a parameter t^+ and N_O.[6] The signature is then sent back to O along with V's identity certificate $\mathrm{cert}(pk_V)$ and t^+. The value of t^+ reflects the desired token expiration period chosen by V.

After V has been authenticated and its identity is known to O, O then decides according to a given policy, which can be application specific, whether to accept or deny V's request for an attestation token. We denote this procedure as checkPolicy(). It takes t^+ and the identity id_V of V as input. O then generates T as follows: It first fetches the set $\mathcal{C} = \{c_1, \ldots, c_z\}$ of benign software configuration for provers in $\mathcal{N}$. This procedure is denoted by getGoodConfigs(). O then hashes this list into one single value

6 The signature scheme used in this protocol does not have to be an OAS scheme. tokenReq can be based on any other digital signature scheme that exploits on existing Public Key Infrastructure (PKI).

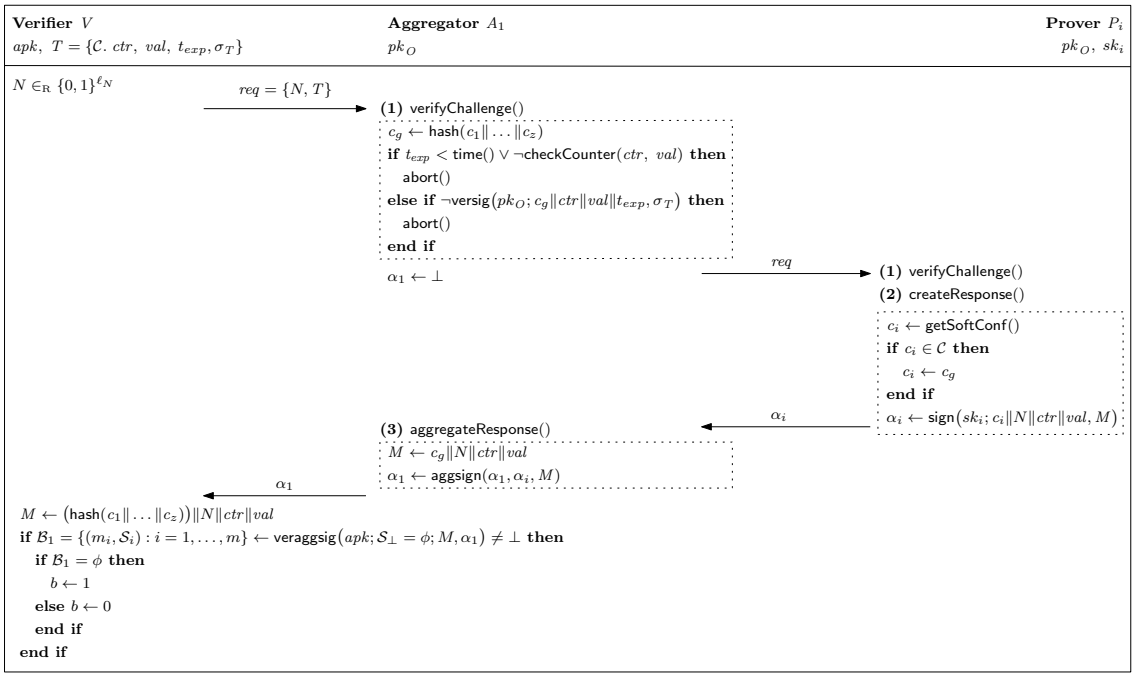


Figure 4.14: Protocol attest

$c_g = \mathsf{hash}(c_1 \| \ldots \| c_z)$, generates the token T, encrypts it, and sends it to V. In particular, V receives from O: an encrypted[7] token ϵ, the aggregate public key *apk* of $\mathcal{N}$, and O's signature σ_O over *apk*. After verifying σ_O, V decrypts T and stores it with *apk*. The token $T = \{\mathcal{C},\ ctr,\ val,\ t_{exp},\ \sigma_T\}$ is formed of the benign configuration set $\mathcal{C}$, the free counter *ctr* and its value *val*, an expiry time t_{exp} (output of the checkPolicy() algorithm), and the signature σ_T. tokenReq is formally:

$$\mathsf{tokenReq}[V : t^+, sk_V; O : sk_O, apk; * : \mathsf{cert}(pk_O), \mathsf{cert}(pk_V)] \rightarrow \left[V : T, apk; O : t_{exp}\right].$$

Attestation: Having a valid attestation token T. a verifier V may attest the network at any time before the token expiry t_{exp} indicated in T. In order to attest $\mathcal{N}$, V picks a random device as initiator D_1, which acts as the root aggregator A_1. A_1 represents the interface for V to attest the network through attest. The details of attest are shown in Figure 4.14. V initiates attest by sending A_1 an attestation request containing the attestation token T and a fresh nonce N. The root aggregator A_1 verifies the request by: checking the counter value in T using checkCounter(), and verifying the signature σ_T included in T using the public key of O. This procedure is referred to as verifyChallenge(). A_1 forwards the attestation request to its neighbors only if verifyChallenge() was successful. The request is then verified and forwarded by every aggregator A_i in $\mathcal{N}$ until it received by every prover P_i in $\mathcal{N}$. As already discussed in Section 4.1.2, this operation leads to the formation of a spanning tree that has provers as leaf nodes, aggregators as intermediaries, and is rooted at A_1.

7 T is encrypted based on V's public key pk_V

Next, every prover P_i measures it software state generating the software configuration c_i, which is then matched within the set of benign software configurations. If $c_i \in \mathcal{C}$, the attestation report of A_i is generated as an OAS signature α_i based on P_i's secret key sk_i over the nonce N, the counter id *ctr* and its value *val*, and the hash c_g of all benign software configuration from T. If $c_i \notin \mathcal{C}$, A_i generates the response as α_i over its actual software configuration c_i. A_i then sends α_i to its parent in the spanning tree. This procedure is referred to as createResponse().

When an aggregator A_j receives attestation responses from its children in the tree, it aggregates them into one OAS signature α_j which is created using aggsign (see Definition 4.2). Note that, aggsign takes $M = c_g \| N \| ctr \| val$ as the default message. A_j then sends α_j to its parent, which aggregates it with other received signatures and sends it to its parent and so on. This procedure is denoted by aggregateResponse(). As a consequence, the provers' attestation reports are propagated and aggregated along the spanning tree until they reach the root aggregator A_1, which creates and sends the aggregated signature α_1 of all provers in $\mathcal{N}$ to V.

V then uses veraggsign to verify α_1 (see Definition 4.2). If veraggsign outputs $\mathcal{B}_1 = \phi$, V deduces that all provers in $\mathcal{N}$ have a good software configuration from $\mathcal{C}$ and outputs $b = 1$. Otherwise, if the output of veraggsign is $\mathcal{B}_1 \neq \phi$ V outputs $b = 0$. In this case, V can extract from $\mathcal{B}_1$ the exact software configuration and identity of all proves that do not have a good software configuration. attest fails when veraggsign fails. This protocol is formed of two sub-protocols, which are formally:

$$\mathsf{attest}_1 \left[V : T, apk; A_1 : -; * : pk_O \right] \rightarrow [V : b; A_1 : T, N] .$$

and:

$$\mathsf{attest}_2 \left[A_i : T, N; D_j : (sk_j); * : pk_O \right] \rightarrow \left[A_i : \alpha_j; D_j : T, N \right] .$$

4.2.4 *Implementation*

We also present two instantiations of this solution on top of SMART [52] and TrustLite [84] security architectures (see Chapter 3). Recall that, these architectures provide their security guarantees based on minimal hardware features. The main features incorporated are a small amount of Read-Only Memory (ROM) and a simple Memory Protection Unit (MPU). In the following we present the two instantiations:

Implementation on SMART. Our implementation on SMART [52] requires the same modifications to SMART's architecture presented in Section 4.1.3, i.e., the MPU is extended to control access to a small amount of rewritable memory. We mainly use this memory to store the list of counters required for mitigating DoS attacks on attestation. We store in ROM of every device the program code responsible for executing all components of attest on a prover, i.e., verifyChallenge() and createResponse(). The ROM also stores for each prover P_i the OAS secret key sk_i. Consequently, integrity of all code and the OAS key is ensured via emutability of ROM. Further, we store the list of counters $ctr_1, \ldots, ctr_s$ and their values $val_1, \ldots, val_s$ in rewritable memory of each prover

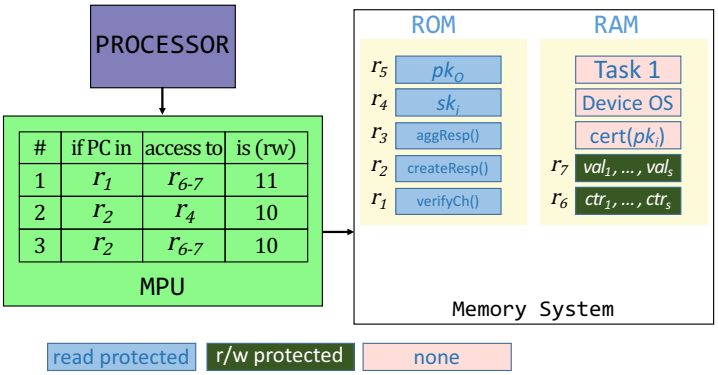


Figure 4.15: Implementation based on SMART [52]

P_i. Note that, this list have to be updated during P_i's lifetime. The implementation on SMART is shown in Figure 4.15 where we denote by RAM the rewritable memory region. SMART's MPU is configured such that it ensures data of attest are only accessible to the unmodified part of its code that requires access to this data. For example rule #2 ensures that createResponse() has read access to sk_i, and rule #1 and #3 ensure that only verifyChallenge() has write access to the counters $ctr_1, \ldots, ctr_s$ and their values $val_1, \ldots, val_s$, which are read accessible to verifyChallenge() and createResponse().

Implementation on TrustLite. Our solution is also implemented as trustlets on TrustLite [84] (see Chapter 3). More precisely, we implemented the code responsible for executing each component of attest on a prover, i.e., verifyChallenge() and createResponse(), as a single independent trustlet. Our implementation is shown in Figure 4.16. Trustlite ensures the software integrity of each of the components through the secure boot component SecureBoot on P_i. Further, as in SMART, the MPU of TrustLite was configured such that it ensures data of attest is only accessible to appropriate trustlets. For example rule #1 ensures that only SecureBoot has read access to the memory storing the program code of attest. rule #3 ensures that only createResponse() has read access to sk_i, and rule #2 and #4 ensure that only verifyChallenge() has write access to the counters $ctr_1, \ldots, ctr_s$ and their values $val_1, \ldots, val_s$, which are read accessible to verifyChallenge() and createResponse().

Implementation of OAS. Our instantiation of OAS from pairings was implemented for the two security architecture as well as other commodity hardware. For this implementation we used the library from [145] that provides efficient cryptographic operations based on pairing, which are suitable for the low-end devices that we target. Finally, we defined the operations of OAS over the pairing friendly elliptic curve BN254 [145]. The security level provided by BN254 is 128 bits.

4.2.5 *Performance Evaluation*

We assess performance of this solution in terms of computational, communication, memory, runtime, and energy costs. Moreover, we also present simulations results for networks of up to 1,000,000 devices as in Section 4.1.4. This performance evaluation is

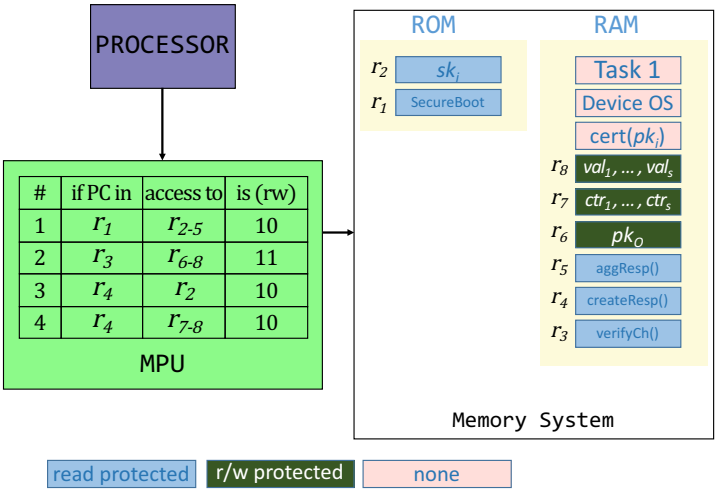


Figure 4.16: Implementation based on TrustLite [84]

based on the implementation from Section 4.2.4. Note that, we also assume here that the topology of the network does not change while the attestation protocol is executing.

Computation Cost. Cryptographic operations, such as generation of the hash c_g of good software configurations or of an OAS signature, constitute the major part of the computation cost. Let g_i denote the number of neighbors of every aggregator A_i, and $h_i \leqslant g_i - 1$ be the upper bound on the number of children of A_i in the spanning tree. The root aggregator A_1, which is chosen by the verifier V to be the interface to $\mathcal{N}$, aggregates up to g_1 OAS signatures and generates only 1 hash. Each aggregator D_i aggregates up to h_i OAS signatures while also generating 1 hash only. Finally, every prover P_i generates 1 hash and 1 OAS signature.

Communication Cost. We used ECDSA as our digital signature scheme, i.e., $\ell_{\text{sign}} = 320$. The signature size of our implementation of OAS is $\ell_{\text{sign}} = 256$. We further used 64 bit for counter values, and chose $\ell_N = 160$. As a consequence, nonces and software configurations are 20 Bytes each. Counters are 2 Bytes and their values are 8 Bytes each. Digital signatures are 40 Bytes each. And, OAS signatures and keys are 32 Bytes each. Let u be the number of good software configurations in $\mathcal{C}$, v be the number of software configurations $c_1, \ldots, c_v$ in $\mathcal{N}$ that do not belong to $\mathcal{C}$, and w be the number of provers having one of the configurations not in $\mathcal{C}$. The size of a token is $20u + 58$ Bytes, while the size of an attestation report containing an aggregate signature is $32 + 32w + 20v$ Bytes. Consequently, each aggregator has a communication overhead, which is upper bounded by receiving $20u + 78 + 32g_i + 32w + 20v$ Bytes and sending $32 + (20u + 78)g_i + 32w + 20v$ Bytes The upper bound on the communication overhead of each prover P_i is receiving $20u + 78$ Bytes and sending 84 Bytes.

Memory Cost. Every prover P_i in $\mathcal{N}$ should store the following: (1) an OAS key pair (sk_i, pk_i) and the corresponding identity certificate cert(pk_i); (2) the list of s counters $ctr_1, \ldots, ctr_s$ and their values $val_1, \ldots, val_s$; and (3) the public key pk_O of $\mathcal{N}$'s operator O. The memory cost of A_i is around $10s + 228$ Bytes, where s is a constant protocol

parameter. Low-end embedded devices, which we target in this solution, have more than 1,024 Bytes of Flash memory. Our solution required less than 34% of a that memory on every prover in order to allow 12 verifiers to execute attestation in the same time interval, i.e., by setting $s = 12$.

Runtime. Similar to Section 4.1, the design of this solution allow devices at the same level of the spanning tree to perform their execution of the attestation protocol at the same time, i.e., in parallel. However, the aggregation of OAS signatures on level l of the tree requires the generation of these signatures by level $l-1$. The overall runtime is also affected by the fan-out of the tree, i.e., the number of children per device, which increases the aggregation time on individual aggregators. Moreover, this solution optimizes the overall communication overhead, such that it is constant if all provers have good software configuration. Consequently, runtime of the attestation protocol is dependent on the overall height $d = f(n) \in \mathcal{O}(\log(n))$ of the spanning tree, the number of provers that do not have a good software configuration, and the number of children per aggregator.

We denote by t_{sign}, $t_{aggsign}$, t_{versig}, t_{hash}, and t_{tr} be the time that a device D_i requires to perform operation sign of OAS scheme, to execute aggsign aggregating two OAS signatures, to verify a digital signature, to generate the hash c_g of all good software configuration in $\mathcal{N}$, and to communicate 1 Byte to a neighboring device D_j respectively. The overall runtime of the attestation protocol is:

$$t \leqslant \left[110l + \sum_{i=0}^{d} (32w_i + 20v_i)\right] \cdot t_{tr} + \left(\sum_{i=0}^{l} h_i\right) \cdot t_{aggsign} + d \cdot (t_{versig} + t_{hash}) + t_{sign}$$

Where v_i denotes the number of software configurations that do not belong to $\mathcal{C}$ in the subtree rooted at aggregator A_i.

In Table 4.1, we show the runtime of all cryptographic operations we used in our solution for TrustLite [84] and a t2.micro Amazon EC2 instance [6].[8] The runtimes shown are in ms and are the average of 100 executions. Empty cells reflect computations that are not performed on the given platform. Further, we show in Table 4.2 the evaluation results the OAS instantiation. The table shows the runtime for each of the algorithms defined in Section 4.2.2.2 in terms of the number n of provers in $\mathcal{N}$, v of software configurations not in $\mathcal{C}$, and h_i of children for each aggregator A_i.

Energy Cost. We denote by $E_{aggsign}$, E_{sign}, E_{versig}, E_{recv}, E_{send}, and E_{hash} the energy that a device D_i needs for aggregating two OAS signatures, generating an OAS signature, verifying a digital signature, receiving 1 Byte, and sending 1 Byte respectively. During one execution of the attestation protocol, the energy $E(A_i)$ consumed by each aggregator A_i can be estimated as:

8 Amazon EC2 we used has 1 GByte of RAM, a 3.3 GHz Intel Xeon Processor, and is running Ubuntu server 14.04.

Table 4.1: Performance of cryptographic functions

Function	**TrustLite** [84] **Run-time (ms)**	**EC2 t2.micro** [6] **Run-time (ms)**
$H : \{0,1\}^l \to \mathbb{G}_1$	921.52	3.39
$g^x,\ g \in \mathbb{G}_1$	1282.71	4.71
$g^x,\ g \in \mathbb{G}_2$	—	11.60
$ab,\ a, b \in \mathbb{G}_1$	86.48	0.32
$ab,\ a, b \in \mathbb{G}_2$	—	0.33
$ab,\ a, b \in \mathbb{G}_T$	—	0.07
$e : \mathbb{G}_1 \cdot \mathbb{G}_2 \to \mathbb{G}_T$	—	7.67

Table 4.2: Performance of OAS algorithms

Function	**TrustLite** [84] **Run-time (ms)**	**EC2 t2.micro** [6] **Run-time (ms)**
sign	2204.23	8.1
genkeysign	—	11.60
aggPK	—	$0.33 \cdot n$
aggsign	$86.48 \cdot h_i$	$0.32 \cdot h_i$
veraggsign	—	$0.33 \cdot \sum_2^v (\lvert S_i \rvert - 1) + (8.16) \cdot v$

$$\begin{aligned} E(A_i) \leqslant & \left(32 + (20u + 78)g_i + 32w + 20v\right) \cdot E_{\mathsf{send}} + \\ & + \left(20u + 78 + 32g_i + 32w + 20v\right) \cdot E_{\mathsf{recv}} + \\ & + h_i \cdot E_{\mathsf{aggsign}} + E_{\mathsf{versig}} + E_{\mathsf{hash}} \end{aligned}$$

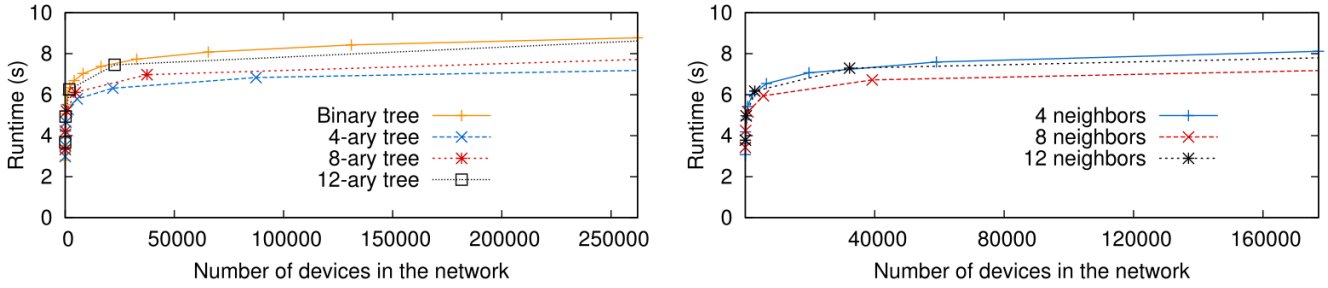
Further, the energy $E(P_i)$ consumed by every prover P_i in $\mathcal{N}$ can be estimated as:

$$E(P_i) \leqslant 84 \cdot E_{\mathsf{send}} + (20u + 78) \cdot E_{\mathsf{recv}} + E_{\mathsf{sign}} + E_{\mathsf{hash}}$$

Simulation Results. We used the OMNeT++ [104] network simulator to assess the performance of this solution for very large networks of embedded devices. The attestation protocol was again implemented on the application layer, where cryptographic operation were emulated with delays corresponding to real measurments of their execution time on TrustLite [84]. For our simulations, the average communication rate between devices was set to 250 Kbps, which corresponds to the defined bandwidth for ZigBee [135]. We considered four topologies for evaluating this solution: a star topology, a chain topology, networks with fixed number of neighbors per device, and tree topologies with num-

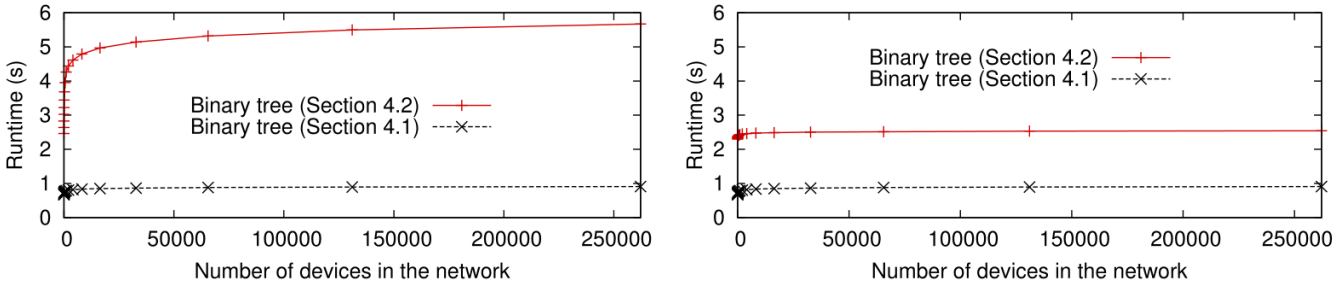
ber of child nodes varying from 2 to 12. Further, we simulated various network sizes, which ranged from 10 to 1,000,000. Figure 4.17 shows the simulation results for tree topologies and networks with fixed number of neighbors. Results for star and chain topologies are omitted as these topologies are not considered an interesting application scenario for collective attestation.

We also compared this solution to the one presented in Section 4.1. We carried out the comparison via simulations in two different scenarios. In the first scenarios we considered every device in $\mathcal{N}$ to be a low-end prover device, i.e., a device with TrustLite architecture that should be attested. The second scenario corresponds to the targeted setting of this solution. In this scenario we assume that aggregators reside on more powerful devices that have a faster intercommunication rate. In particular, the aggregators in $\mathcal{N}$ were composed of 20% t2.micro EC2 instances [6], 30% Intel Galileo[9] devices, and 50% Raspberry Pi[10] devices. The results of these simulations are shown in Figure 4.18.



(a) Tree topologies (b) Fixed number of neighbors

Figure 4.17: Performance of attestation



(a) First scenario (b) Second scenario

Figure 4.18: Comparison to the other solution

The results of our simulations show that, for a fixed number of provers with malicious software, the runtime of attestation is logarithmic in the size of the network for both tree topologies (Figure 4.17a) and networks where devices have a fixed number of neighbors (Figure 4.17b).

9 Intel Galileo is equipped with a 256 MBytes of RAM and a 400 MHz CPU.
10 Raspberry Pi is equipped with a 512 MBytes of RAM and a 700 MHz CPU.

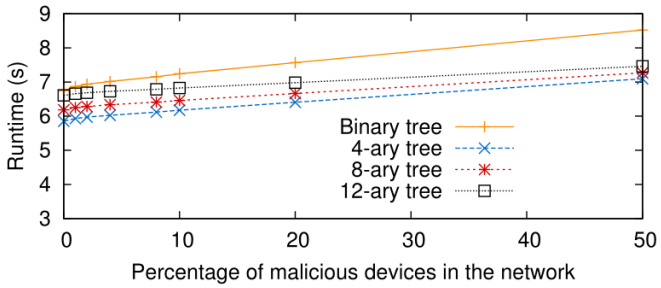


Figure 4.19: Performance of attestation as function of the number of malicious provers

The comparison in Figure 4.18a shows that the solution from Section 4.1 performs better compared to the solution we present in this section, when all devices in the network are low-end embedded device communicating over ZigBee. However, the solution we present in this section provides more flexibility as its does not impose any constraints on the devices that act as aggregators. In particular, aggregators do not have to be on low-end IoT devices equipped with lightweight security architecture. Devices acting as aggregators can be completely untrusted network devices, e.g., cloud servers or routers. Further, this solution also provides better resiliency against physical attacks, which allows its applicability to a wider range of applications.

Therefore, for a fair comparison, we present in Figure 4.18b the runtimes of the two solutions in a more realistic scenario. The figure shows that for very large deployments the runtime difference between the two solutions is as low as 1.5 seconds.

In conclusion, our evaluation shows that the solution presented in this chapter performs *in its targeted settings* almost as good as the solution presented in Section 4.1, while providing DoS mitigation, resiliency to physical attacks, and reporting the software configurations of devices that failed attestation. The performance of this solution may also leverage hardware acceleration for the signature computation, aggregation, and verification.

Finally, we show in Figure 4.19 the runtime of attestation as a function of the percentage of malicious provers. The results show that the runtime is linear in tree topologies in the number of malicious provers. This is a direct consequence of the fact that attestation responses include the public keys of all malicious provers. Therefore, the communication overhead is linear in the number of those devices. The figure demonstrates that the runtime is still acceptable in large networks with $10,000$ devices, even if a large fraction, i.e., up to 50%, of provers have a malicious software configuration.

4.2.6 *Security Analysis*

As in Section 4.1.5, the goal of our attestation solution is to enable the verifier V to check the software integrity of all provers in a network $\mathcal{N}$, i.e., V should return $b = 1$, i.e., accept the attestation report, if every prover P_i in $\mathcal{N}$ has an unmodified software that is accepted by V, e.g., latest version of benign software. This should hold even if all aggregators are under full control of the adversary. Moreover, even though physically

attacked prover may evade their own detection, they should not be able to help other provers evade detection. This security goal can be formalized as a security experiment $\mathbf{Exp}_{\mathcal{A}}$, where the adversary $\mathcal{A}$ can interact with every device in $\mathcal{N}$ as well as V. $\mathcal{A}$ has full control over the communication channel between every two devices in $\mathcal{N}$, and between $\mathcal{N}$ and V, i.e., it can eavesdrop on, modify, drop, and inject arbitrary messages to any $D_i \in \mathcal{N}$ and V. $\mathcal{A}$ exploits physical proximity to tamper with the hardware of a small number of provers $P_1, \ldots P_p$, and some (or all) aggregators $A_1, \ldots A_q$. It also maliciously modifies the software configuration of one (or more) provers P_c without tampering with their hardware. At the end of the experiment, V outputs the result of attestation b indicating whether it has accepted the attestation report or not, following a polynomial number (in ℓ_N and ℓ_{sign}) of steps performed by $\mathcal{A}$. The output of V represents the result of this security experiment, i.e., $\mathbf{Exp}_{\mathcal{A}} = b$. In the following we provide the definition of secure collective attestation under the pre-described adversary model:

Definition 4.5 (Secure collective attestation under physical attacks)**.** *Let* f *be a polynomial function in* ℓ_N *and* ℓ_{sign}*. We consider a collective attestation scheme to be secure under physical attacks if the probability* $\Pr\left[b = 1 | \boldsymbol{Exp}_{\mathcal{A}}(1^{\ell}) = b\right]$ *is negligible in* $\ell = f(\ell_N, \ell_{\mathsf{sign}})$*.*

Theorem 4.2 (Security of our attestation solution)**.** *The attestation solution presented in this chapter is a secure collective attestation scheme (Definition 4.5) if the underlying digital signature scheme is selective forgery resistant, and the OAS scheme is unforgeable according to Definition 4.4.*

Proof sketch of Theorem 4.2. Let *apk* be the aggregate OAS key of $\mathcal{N}$, and let the default message for attestation be $M = \big(\mathsf{hash}(c_1 \| \ldots \| c_z) \| N \| ctr \| val\big)$, where $c_1, \ldots, c_z$ are the good software configurations in $\mathcal{C}$. As a response to a challenge containing the nonce N and a token T including a counter *ctr* with value *val*, the verifier V receives from the root aggregator an attestation report formed of the OAS signature α_1. V accepts the attestation report and returns $b = 1$ only if $\mathcal{B} = \{(m_1, S_1), \ldots, (m_\nu, S_\nu)\} = \mathsf{veraggsign}(apk, S_\perp, M, \alpha_1)$ such that $S_\perp = \phi$, and $\mathcal{B} = \phi$. Consider the following five strategies through which the $\mathcal{A}$ tries to cover compromise of prover P_c: (1) $\mathcal{A}$ keeps P_c's report α_c unmodified, (2) $\mathcal{A}$ replays an old report of P_c containing an OAS signature α_{old} over the hash c_g of all good software configuration, (3) $\mathcal{A}$ tampers with $\mathcal{C}$ so that it includes P_c's malicious software configuration, (4) $\mathcal{A}$ forges an OAS signature which associates a signature over $M = \big(\mathsf{hash}(c_1 \| \ldots \| c_z) \| N \| ctr \| val\big)$ to P_c; or (5) $\mathcal{A}$ tampers with the aggregate public key *apk* used by V to verify the final attestation report.

We first consider the case where $\mathcal{A}$ keeps α_c unmodified . Due to hardware security architecture on devices acting as provers, $\mathcal{A}$ is not capable of tampering with the code of attest responsible for measurement and reporting of P_c's software or extracting its OAS secret key sk_c. Consequently, the modified software configuration c'_c will be included in the OAS signature α_c generated by P_c as c'_c instead of c_g, if the integrity measurement responsible for the generation of c'_c is capable of detecting this modification, i.e., if $c'_c \notin \mathcal{C}$. In this case, V will always return $b = 0$. On the other hand, when replaying an old report containing the signature α_{old} for c_g, the signature will contain an old nonce N_{old}. Consequently, V will only accept and return $b = 1$ if $N_{old} = N$ which is negligible in ℓ_N.

We now consider the case where $\mathcal{A}$ tries tampering with $\mathcal{C}$ to include c'_c. In order to achieve this, $\mathcal{A}$ must forge the digital signature σ_O, which is generated by O over the token T. This strategy will be successful with a probability that is negligible in ℓ_{sign}, here ℓ_{sign} is the security parameter of the digital signature scheme used to sign T. Additionally, since $\mathcal{C}$ is also authenticated with α_c in the attestation report of P_c, tampering with $\mathcal{C}$ will change α_c and cause V to return $b = 0$.

Next we consider the case where $\mathcal{A}$ tries to generate an OAS signature that associates a signature on $M = \big(\mathsf{hash}(c_1 \| \ldots \| c_z) \| N \| ctr \| val\big)$ to P_c. However, P_c does not generate such a message since $c'_c \notin \mathcal{C}$, and N is a newly generated nonce. Consequently, according to the unforgeability of OAS defined in Definition 4.4, $\mathcal{A}$ can find such a signature with a probability that is negligible in ℓ_{sign}, where ℓ_{sign} is the security parameter of the OAS scheme used to sign attestation reports.

Finally, if $\mathcal{A}$ was capable of tampering with the *apk* to replace the public key pk_c with its own public key $pk_{\mathcal{A}}$, $\mathcal{A}$ would then be able to sign any message on behalf of P_c, i.e., $\mathcal{A}$ would be able to generate an attestation report for P_c which contains an OAS signature over $M = \big(\mathsf{hash}(c_1 \| \ldots \| c_z) \| N \| ctr \| val\big)$ independent of the software configuration of P_c. The success probability of this strategy is negligible in ℓ_N and the security parameter ℓ_{sign} of the digital signature that is used to protect the integrity of *apk* between V and O.

Therefore, the probability of $\mathcal{A}$ convincing V to accept the attestation report and return $b = 1$ after maliciously modifying the software of at least one prover in $\mathcal{N}$ is negligible is negligible in ℓ_N, and ℓ_{sign}. This result is independent of the number of aggregators or provers (other than P_c) that are physically attacked by $\mathcal{A}$.

□

4.2.7 *Threshold Attestation*

As shown in Section 4.2.5, the runtime of our solution is constant in the network size, however, it increases linearly with the number of malicious provers in $\mathcal{N}$. Our ultimate goal is to provide a collective attestation solution for networks of embedded devices which is constant time. The solution should allow a low-end verifier to efficiently attest a network of embedded devices, regardless of the number or condition of devices in this network. We now present an simple extension to our solution that enables preserving the constant time property independent of the number of provers with malicious software configurations.

The intuition behind this extension is the following: Although there might be in some applications a linear correlation between number of provers with malicious software and the overall network size, i.e., a given percentage of provers in $\mathcal{N}$ are compromised, the quantity of tolerated compromise is however usually limited. This is reasonable to assume as the tolerated compromise is usually dependent on the amount redundancy rather than number of devices in $\mathcal{N}$. Therefore, one can limit the increase in runtime by simply setting an upper bound (i.e., a threshold) on the number of compromised devices to be identified by V.

The upper bound is determined by V, and is sent to O during tokenReq protocol in order to be embedded in the token T. During attestation, aggregators use T to determine

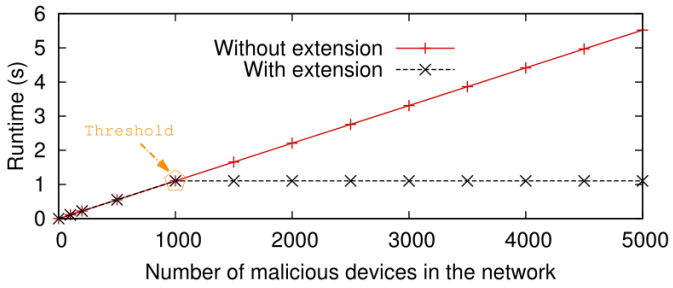


Figure 4.20: Verification of threshold attestation using an EC2 t2.micro verifier

this upper bound. They then stop aggregating OAS signatures over malicious software whenever this upper bound is crossed. All signatures received after the upper bound has been crossed are dropped. This will lead to the generation of final attestation report that is not verifiable using the aggregate public key *apk* known to V. Consequently, whenever V fails to verify the attestation report, it learns that the number of malicious provers in $\mathcal{N}$ might have exceeded the accepted upper bound.

Note that, other factors could cause the verification of the attestation response to failure, e.g., DoS attacks or communication errors. Therefore, it is crucial to allow V to ensure that verification failure was caused by large number of malicious provers that exceeded the upper bound. A possible way to achieve this is by dividing the attestation report in two parts: The first part is formed of a multisignature over the default message allowing V to attest provers with good software configuration. The second part is an aggregate signature over all malicious software configuration. This allows V to make sure that the upper bound for malicious devices has been exceeded by verifying the aggregate signature in constant time. By doing so, V also learns the identities of malicious provers as well as their software configuration. Figure 4.20 shows how threshold attestation achieves a constant time verification time. The threshold in the figure is set to 1000 malicious provers in a network with $10,000$ provers and a fixed number 4 of neighbors per device.

4.2.8 *Conclusion*

In this section we presented a second solution for efficient collective attestation. However, unlike Section 4.1, this solution provides stronger security guarantees, imposes less assumptions, and is applicable to a wide range of applications involving devices that are not trusted by the same entity. These benefits came at the cost of an additional overhead caused by the use of a novel signature scheme. This scheme provides scalability and public verifiability, and allows untrusted intermediaries. It is important to stress that the two solutions presented in the previous and the current section complete each other. Section 4.1 presents a solution that has best achievable efficiency and is applicable in certain scenarios, while the solution presented in this section allows wider applicability and provides stronger security guarantees by imposing minimal additional overhead. Note that, these solutions are based on identifying security requirements in existing

applications and designing solutions that satisfy these requirements. In Section 4.3 we try to establish collective attestation on solid grounds by systematically analyzing its requirements, components, and properties.

4.3 SYSTEMATIC TREATMENT OF COLLECTIVE ATTESTATION

So far, we have presented two collective attestation solutions for large networks of embedded devices. The two solutions complete each other, as they are applicable in different scenarios and have different requirements and security guarantees. However, these solutions are created in an ad-hoc fashion. They are based on identifying the problem setting and requirements in a specific use case scenario, and devising the appropriate solution that is specifically tailored to satisfy these requirements in the specified setting.

In this section we aim at providing a generic solution for the problem of malware infestation that is not dependent on a well defined use case. First, we present the most realistic security model for collective attestation, outlining the different adversarial capabilities and presenting various adversarial classes. We then provide a systematic treatment of collective attestation, which establishes the problem of detecting malware infestation in large networks on solid grounds, and represents a guide in this area. In particular, we present a careful analysis of the security properties that should be provided by a secure collective attestation solution. And, we identify the requirements in terms of hardware, software, and protocol features, that allow satisfying these properties. Finally, we analyze the collective attestation solutions proposed in Section 4.1 and 4.2 in light of the identified properties, and investigate their security under a stronger adversary model.

Contribution. We investigate the problem of detecting malware infestation in large embedded networks and provide the first systematic treatment of collective attestation which includes: (1) identifying the threat model, which is substantially different than the standard adversary model assumed in all existing single device attestation protocols; (2) extracting the security goals, properties, and requirements of a collective attestation solution; and (3) determining the features in terms of protocol, software, and hardware components that should be maintained by a collective attestation solution. Further, we evaluate the security of the collective attestation solutions presented earlier in this chapter, and provide various attacks that could be launched on them by a more powerful adversary. Moreover, we devise a generic collective attestation solution which satisfies the identified requirements and properties, and is secure in the presence of our realistic adversary. Finally, we show how to instantiate the generic solution on recent security architectures for low-end embedded devices with different security features and functional capabilities, i.e., SMART [52] and TrustLite [84], and present extensive performance evaluation based on these two instantiations, in addition to simulations of the solution in networks of up to $1,000,000$ devices.

Outline. After introducing the system and adversary models and defining our security notion in Section 4.3.1, we present the properties required by a secure collective attestation solutions in Section 4.3.2, and extract the features required to achieve these properties in Section 4.3.3. Next, the security of the two collective attestation solutions

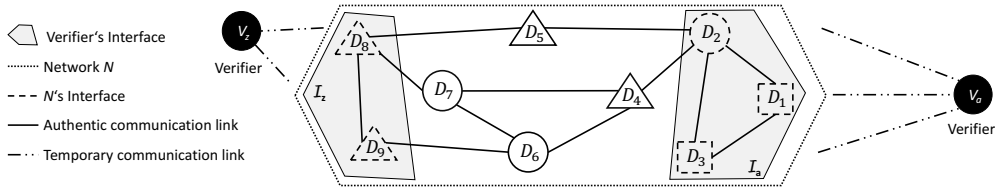


Figure 4.21: Example network of nine members

presented earlier in this chapter is analyzed in Section 4.3.4, and a secure and generic collective attestation solution is presented in Section 4.3.5. We describe our implementation of this solution in Section 4.3.6. Performance evaluation is then presented on Section 4.3.7, and the section concludes in Section 4.3.8.

4.3.1 *Definitions*

4.3.1.1 *Problem Description and System Model*

Network. We consider a network $\mathcal{N} = D_1, \cdots, D_n$ that is formed of n interconnected members D_i collaborating to fulfill a task (see Figure 4.21). The members of $\mathcal{N}$ are usually (low-end or high-end) heterogeneous embedded devices that interact allowing $\mathcal{N}$ to export services to other entities. It is assumed that $\mathcal{N}$'s members are redundant in order to tolerate failures and/or attacks against $\mathcal{N}$. $\mathcal{N}$ may not have a routing protocol in place. However, members of $\mathcal{N}$ should be able to communicate to their direct neighbors [44, 68, 114, 115], exchanging commands or data. A verifier V_a is an entity that is interested in (one or more) services offered by $\mathcal{N}$. V_a assesses the trustworthiness of $\mathcal{N}$ in order to ensure the trustworthiness of the requested service. The set of all verifiers is denoted by $\mathcal{V} = \{V_a, \cdots, V_z\}$. We denote by the interface $\mathcal{I} = D_u, \cdots, D_v$ of $\mathcal{N}$ the set of all members that represent a gateway between $\mathcal{N}$ and the outside world. A specific verifier V_a may interact with $\mathcal{N}$ through multiple members of that set. We refer to these members as V_a's attestation interface $\mathcal{I}_a$ to $\mathcal{N}$. Finally, a trusted network operator O is responsible for the establishment of a secure communication link from any verifier in $\mathcal{V}$ to $\mathcal{N}$. O acts as $\mathcal{N}$'s certification authority. A collective attestation solution should allow any verifier $V_a \in \mathcal{V}$ to verify the trustworthiness of a requested service in an efficient and scalable manner.

Members. Members of $\mathcal{N}$ may be heterogeneous (i.e., have multiple hardware and software configurations). However, we assume that every member D_i of $\mathcal{N}$ satisifes the requirements that allow secure remote attestation as discussed in Section 4.1.1, i.e., D_i should be equipped with a lightweight security architecture that has a minimal amount of Read-Only Memory (ROM) and a simple Memory Protection Unit (MPU), e.g., SMART [52] and TrustLite [84]. We consider two class of members: low-end embedded devices that are constrained in resources, e.g., sensors, and high-end devices with sufficient resources such as a Raspberry Pi. Every member of $\mathcal{N}$ is considered as an independent entity that is capable of performing a specific task, e.g., sensing and/or actuation.

4.3.1.2 *Collective Attestation*

A collective attestation solution is a security protocol involving three different parties: (1) the network $\mathcal{N}$ to be attested; (2) a verifier V_a from the set $\mathcal{V} = \{V_a, \cdots, V_z\}$ of all possible verifiers that is interested in a service offered by $\mathcal{N}$; and (3) the network operator O that is responsible for the establishment of secure communication links within $\mathcal{N}$, and between $\mathcal{N}$ and $\mathcal{V}$. The goal of a secure collective attestation solution is to enable every $V_a \in \mathcal{V}$ to verify the trustworthiness of any service delivered by $\mathcal{N}$. The trustworthiness of the service is defined as the trustworthiness of $\mathcal{N}$, which implies the software integrity of every member D_i in $\mathcal{N}$. Consequently, a secure collective attestation solution should ensure V_a that the software state of every member D_i in $\mathcal{N}$ has not been maliciously modified.

Definition 4.6 (Collective Attestation)**.** *A secure collective attestation solution comprises the following six protocols / functions:*

- $\mathsf{setup}(\cdot)$*: This protocol is executed between O from one side, and $\mathcal{N}$ and $\mathcal{V}$ from the other side. It allows secure key provisioning by O for $\mathcal{V}$ and $\mathcal{N}$, which in turn enables secure interactions between these entities. Consequently,* $\mathsf{setup}(\cdot)$ *allows each member D_i of $\mathcal{N}$ to obtain: (1) the cryptographic key(s) k_i that allows D_i to authenticate messages exchanged with every other member $D_j \in \mathcal{N}$ and verifier $V_a \in \mathcal{V}$; and (2) a token h_i that enables D_i to announce its reference software state to every other member $D_j \in \mathcal{N}$ and a verifier $V_a \in \mathcal{V}$. Similarly, every verifier $V_a \in \mathcal{V}$ acquires the cryptographic key(s) K necessary to authenticate $\mathcal{N}$, and a token H for verifying $\mathcal{N}$'s state.*

- $\mathsf{init}(\cdot)$*: This protocol allows the initiation of collective attestation. It is executed between a verifier $V_a \in \mathcal{V}$ and the member(s) V_a's attestation interface $\mathcal{I}_a$ to $\mathcal{N}$. Consequently,* $\mathsf{init}(\cdot)$ *allows each member D_i of $\mathcal{N}$ to obtain an attestation challenge from V_a.*

- $\mathsf{attest}(k, \cdot)$*: This represents a deterministic function executed on every member D_i of $\mathcal{N}$. Having the cryptographic key(s) k_i of D_i as input,* $\mathsf{attest}(k, \cdot)$ *outputs the attestation response θ_i which reflects its current software state.*

- $\mathsf{interact}(k, \cdot, \cdot)$*: This protocol allows the accumulation/aggregation of attestation responses. It is executed between neighboring members of $\mathcal{N}$, i.e., members that have a direct communication link. The protocol is based on the token h_i, the cryptographic key(s) k_i, and the response θ_i of each involved member D_i. Consequently,* $\mathsf{interact}(k, \cdot, \cdot)$ *allows the member(s) of $\mathcal{I}_a$ to obtain the global attestation response(s) Θ of $\mathcal{N}$, and every other member D_i of $\mathcal{N}$ to obtain an intermediate attestation response ϑ_j.*

- $\mathsf{report}(k, \cdot)$*: This protocol allows the verifier V_a to obtain the global attestation response(s) Θ of $\mathcal{N}$ from the member(s) of $\mathcal{I}_a$.*

- $\mathsf{verify}(K, \cdot, \cdot)$*: This represents a deterministic function executed on the verifier V_a . Having the cryptographic key(s) K, the token H of V_a, and the global attestation response(s) Θ of $\mathcal{N}$ as input,* $\mathsf{verify}(K, \cdot, \cdot)$ *determines whether $\mathcal{N}$ is in a trustworthy state.*

The software state of a member $D_i \in \mathcal{N}$ is reflected in the attestation response θ_i of D_i, and the software state of all $\mathcal{N}$'s members is reflected in Θ.

4.3.1.3 *Threat Model*

We now present the most realistic adversary model in the context of large embedded networks. Indeed, our adversary model covers the standard adversary for remote attestation that is only capable of software attacks. This model is extended with more capabilities and strategies that are applicable in the context of large embedded networks. Based on adversarial capabilities, we consider four kinds of adversaries targeting the devices in $\mathcal{N}$. In the following we present these adversaries:

Software Adversary. A software adversary is the standard adversary in all prior attestation protocols including the collective attestation solution presented in Section 4.1. $\mathcal{A}$ is capable of exploiting software vulnerabilities to compromise the software of any member D_i of $\mathcal{N}$. Compromising the software of D_i allows $\mathcal{A}$ to execute malicious functionality and extract the secrets of D_i that are not protected by hardware. $\mathcal{A}$ may compromise the software of an unlimited number of members of $\mathcal{N}$. In fact, the adversary can exploit a common vulnerability to compromise the software of all members that run the same software.

Physical Adversary. A physical adversary is not considered in all prior attestation protocols including the collective attestation solution presented in Section 4.1. $\mathcal{A}$ has physical access, and is hereby capable of capturing and physically attacking the hardware of some members of $\mathcal{N}$. Manipulating with a device's hardware allows $\mathcal{A}$ to execute malicious functionality, and read/write to memory regions that are protected by hardware. Furthermore, a physical adversary is capable of cloning physically attacked members, and inserting these clones into the network. Physical attacks are indeed not scalable, and $\mathcal{A}$ is limited to physically attacking a small number of $\mathcal{N}$'s members.

Network Adversary. A network adversary has complete control over the communication channel, i.e., it can eavesdrop on, modify, drop, or replay any message exchanged between all members in $\mathcal{N}$ and between any member D_i of the interface $\mathcal{I}_a$ and the verifier V_a.

Mobile Adversary. A mobile adversary has the same capabilities of a software adversary. However, it follows a distinct strategies for evading detection by an attestation protocol which is mostly relevant for embedded networks. An important property of $\mathcal{A}$ is that it is capable erasing all traces of software compromise of any member D_i of $\mathcal{N}$, by restoring D_i's software to its genuine state. We distinguish between two actions that can be performed by $\mathcal{A}$:

- $\mathsf{infect}(D_i)$: This action allows $\mathcal{A}$ to compromise the software of a member D_i of $\mathcal{N}$.
- $\mathsf{disinfect}(D_i)$: This action allows $\mathcal{A}$ to restore the software of a member D_i to its genuine state.

We describe two important strategies that could be followed by $\mathcal{A}$ in order to undermine the security of a collective attestation solution, and evade the detection of software compromise.

- **TOCTOU:** In a Time of Check Time of Use (TOCTOU) attack, $\mathcal{A}$ identifies the members $D_i, \cdots, D_j$ of $\mathcal{N}$ whose software should be compromised in order to manipulate a service offered by $\mathcal{N}$ to the verifier V_a. Next, $\mathcal{A}$ utilizes $\mathsf{infect}(D_i), \cdots, \mathsf{infect}(D_j)$ to compromise the software of these members in the time between the execution of the collective attestation solutions by V_a and the delivery of the service by $\mathcal{N}$. This attack may require utilizing $\mathsf{disinfect}(D_i), \cdots, \mathsf{disinfect}(D_j)$ to disinfect the members $D_i, \cdots, D_j$ if software compromise is done before the execution of the collective attestation solution.

- **Hide & Seek:** In a Hide & Seek attack, $\mathcal{A}$ simply moves between members of $\mathcal{N}$ during the execution of the collective attestation solution in order to evade detection. In particular, while the attestation protocol is being executed, $\mathcal{A}$ utilizes $\mathsf{infect}(D_i), \cdots, \mathsf{infect}(D_j)$ to compromise the software of members $D_i, \cdots, D_j$ of $\mathcal{N}$ that are not yet attested. Next, when those members are to be attested, $\mathcal{A}$ utilizes $\mathsf{disinfect}(D_i), \cdots, \mathsf{disinfect}(D_j)$ to restore their software to a genuine state, while compromising the software of other members $D_k, \cdots, D_l$ that have already been attested.

Finally, we present the game between a verifier V_a and a network $\mathcal{N}$ that allows us to define the security of a collective attestation solution:

Game 1 (Collective attestation forgery)**.** *The interaction between the verifier V_a and the network $\mathcal{N}$ is executed as follows:*

1. *This interaction is initiated by V_a which executes* $\mathsf{init}(\cdot)$ *with $\mathcal{N}$, i.e., V_a generates and sends to member(s) D_i of its attestation interface $\mathcal{I}_a$ the challenge(s) req_i.*

2. *The members of $\mathcal{N}$ have oracle access to the function* $\mathsf{attest}(k, \cdot)$*. This oracle outputs for every member D_i the attestation responses $\{\theta_i^1, \cdots, \theta_i^p\}$ as a response to p attestation queries.*

3. *The members of $\mathcal{N}$ also have oracle access to the protocol* $\mathsf{interact}(k, \cdot, \cdot)$*. This oracle outputs for every member D_i the intermediate attestation responses $\{\vartheta_i^1, \cdots, \vartheta_i^q\}$ as a response to q attestation queries containing attestation responses of other members D_j of $\mathcal{N}$. If D_i is a member of $\mathcal{I}_a$, the oracle alternatively outputs the global attestation responses $\{\Theta^1, \cdots, \Theta^q\}$ of $\mathcal{N}$.*

4. *As a response to V_a's challenge, the member(s) of attestation interface $\mathcal{I}_a$ send the global attestation response Θ to V_a.*

5. *Upon receiving Θ, V_a verifies it by executing* $\mathsf{verify}(K, \cdot)$*.* $\mathsf{verify}(K, \cdot, \cdot)$ *determines whether $\mathcal{N}$ is in a trustworthy state. The output of* $\mathsf{verify}(K, H, \Theta)$ *represents the output of this game.*

Trustworthy members of a network $\mathcal{N}$ utilize $\mathsf{attest}(k, \cdot)$ to create their attestation responses . Moreover, whenever a member D_i interacts with other trustworthy members, it utilizes $\mathsf{interact}(k, \cdot, \cdot)$ to create an intermediate attestation response. Similarly, if $\mathcal{N}$ is trustworthy, the member(s) of the attestation interface $\mathcal{I}_a$ would utilize $\mathsf{interact}(k, \cdot, \cdot)$ to

create a global attestation response(s). On the contrary, a malicious member that desires to cover the compromise of its software or the software of neighboring members, should simulate attest or interact respectively. Similarly, if $\mathcal{N}$ is not trustworthy, the member(s) of ST_a should simulate interact in order to hide this fact from the verifier V_a. According to this game, the security of collective attestation is defined in the following:

Definition 4.7 (Security against collective attestation forgery)**.** *Let $\mathcal{N}$ be any network comprising probabilistic polynomial time members, and let K and k be two sufficiently large values. A collective attestation solution $CA = (\mathsf{setup}, \mathsf{init}, \mathsf{attest}, \mathsf{interact}, \mathsf{report}, \mathsf{verify})$ is considered secure if the probability $\Pr\left[Collective\_Attestation\_Forgery_{V_a,\,\mathcal{N}}(K,k) = 1\right]$ is negligible.*

Theorem 4.3 (Secure collective attestation)**.** *A collective attestation solution is a secure remote attestation protocol for large networks if it secure against Collective_Attestation_Forgery defined in Definition 4.7.*

4.3.2 *Properties of Collective Attestation*

In order to be secure against *Collective_Attestation_Forgery*, a collective attestation solution $CA = (\mathsf{setup}, \mathsf{init}, \mathsf{attest}, \mathsf{interact}, \mathsf{report}, \mathsf{verify})$ should retain a set of properties that protect it against attacks. We start with the general properties required for securing remote attestation. A remote attestation protocol is considered secure, if the component that measures and reports the software state of the prover (i.e., attest) allows secure and accurate measurement and reporting of this state [57]. In particular, a secure remote attestation protocol requires the prover to retain the following properties:

- *Property #1:* Guarantee the secrecy of key(s) k, i.e., attest should not leak any information regarding k, to which it has exclusive access.
- *Property #2:* Ensure immutability of attest's code, i.e., the prover device should prevent modifications to the code of attest which would lead to leaking k or taking incorrect measurements of the prover's software state.
- *Property #3:* Guarantee atomic execution of attest, i.e., the prover device should ensure that attest is executed uninterrupted starting at its very first instruction. Allowing attest to be executed partially would lead to leaking k or taking incorrect measurements of the prover's software state.

A collective attestation solution is secure against *Collective_Attestation_Forgery*, only if every member D_i of $\mathcal{N}$ executing attest satisfy the properties #1, #2, and #3. Additionally, since all these properties are required to guarantee the secrecy of k, every component that has access to k should retain these properties, e.g., interact that performs hop-by-hop aggregation in Section 4.1. Similarly, property #2 and property #3 enable correct measurement of the software state of the prover. Consequently, every component that alters the measurement of the prover's software state should retain these properties, e.g., interact that generates random challenges in Section 4.1.

Having satisfied the properties that enable secure remote attestation, a collective attestation solution would still not be secure against *Collective_Attestation_Forgery*. In particular, we provide multiple attack vectors on collective attestation solutions that satisfy the properties mentioned above.

- **Attack vector I:** The interact protocol could be diverted by a network adversary in order to convince multiple members of $\mathcal{N}$ to incorporate the attestation response of the same member D_i. This attack would lead to overcounting benign devices, thus allowing malicious devices to evade detection.
- **Attack vector II:** The inconsistency of attestation responses, in terms of time of measurement, could allow a mobile adversary to hide its presence in the network following the Hide & Seek strategy. This attack would lead to convincing V_a of the trustworthiness of $\mathcal{N}$, when the software of multiple members could have been compromised.
- **Attack vector III:** The interact protocol could be diverted by a physical adversary that has extracted the secret key(s) k of a member D_i of $\mathcal{N}$. This might allow the adversary to forge the attestation response of the whole network.
- **Attack vector IV:** The time gap between the execution of the collective attestation solution and the actual delivery of $\mathcal{N}$'s service could allow a software adversary to compromise that service after successfully passing the attestation protocol.

Based on these attack vectors, we identified four additional properties that should be satisfied by a collective attestation solution $CA = (\mathsf{setup}, \mathsf{init}, \mathsf{attest}, \mathsf{interact}, \mathsf{report}, \mathsf{verify})$ in order to be secure against *Collective_Attestation_Forgery*. In particular, it is required that the reporting of CA provides correct and consistent attestation responses, where the security of one member is independent on other members, and there exist no exploitable time gap between attestation and service. The required properties are outlined below:

- *Property #4:* The goal of a collective attestation scheme is counting (see Section 4.1) or identifying (see Section 4.2) members of $\mathcal{N}$ whose software is compromised. A secure collective attestation solution should ensure reporting the correct software state of every member D_i in $\mathcal{N}$, i.e., interact should enable a correct one-time integration of every measurement of members of $\mathcal{N}$. This property closes attack vector I.
- *Property #5:* A secure collective attestation solution should guarantee that the global attestation response reflects the state of $\mathcal{N}$ at a specified point in time, i.e., attest should guarantee that the measurements taken at every member of $\mathcal{N}$ are within a negligible time frame. This property is denoted by the *time consistency* property. It closes attack vector II.
- *Property #6:* Since attestation requires hardware features, a physical adversary may allow a member D_i of $\mathcal{N}$ to evade detection by collective attestation through physically attacking its hardware. However, a secure collective attestation solution should not allow a physical attack on a member D_i to affect the outcome of

attestation of another member D_j whose hardware is not physically attacked. This property is denoted by the *hardware independence* property. It closes attack vector III.

- *Property #7:* Attestation is usually accompanied with the request of a service, i.e., the execution of certain pieces of code by various members of $\mathcal{N}$. A secure collective attestation solution should guarantee that the executed code is the same as the attested one, i.e., every member D_i executes the same code it measures through attest. This property is required to close attack vector IV.

4.3.3 *Features of Collective Attestation*

The properties identified in Section 4.3.2 allows securing collective attestation. In this section we derive the features necessary for satisfying these properties, and allowing collective attestation to be secure against *Collective_Attestation_Forgery*. These features are classified into hardware and protocol features.

4.3.3.1 *Hardware Features*

We start with the hardware features that allow every attested member D_i in $\mathcal{N}$ to perform secure remote attestation. In order to statify properties #1, #2, and #3, a device D_i performing attest should be equipped with the following security features in hardware [57]:

- *Hardware feature #1:* A Read-Only Memory (ROM) which ensures the immutability of attest code and prevent the adversary from being able to modify it. The immutability of ROM could be either directly exploited by using it to store the code for attest [52], or indirectly via secure boot [84].
- *Hardware feature #2:* A Memory Protection Unit (MPU) which guarantees the secrecy of the key(s) k by making them accessible only to the code of attest. This kind of access control could be established based on the value of Program Counter (PC) [52, 84].
- *Hardware feature #3:* A mechanism for enabling/disabling interrupts, which prevents the adversary from partially executing the code of attest. This is done by disabling interrupts before attest execution and enabling them after its termination.
- *Hardware feature #4:* A mechanism for enforcing attest to be invoked from its very first instruction, which also prevents adversary from partially executing the code of attest. This can be done through control hardware that only allows jumps to attest code, when the source instruction also belongs to attest code.
- *Hardware feature #5:* A mechanism for securely resetting the device in case of violation of the MPU or atomic execution policies. Secure reset allows erasing execution traces when one of the security policies is breached. Thus, preventing leakage of k.

As already mentioned, these features are also required for other components that access k or influence the generation of the measurement, e.g., interact in Section 4.1. On the other hand, if interact requires no access to k and is not capable of altering the measurement of software states, then members that only execute interact are not required to possess these features, e.g., aggregators in Section 4.2.

On top of these features that allow securing remote attestation, one additional hardware feature is required to enable security against *Collective_Attestation_Forgery* of collective attestation:

- *Hardware feature #6:* A Reliable Read-Only Clock (RROC) which enables each member to securely indicate the time of measurement of software state through attest. RROC denotes a clock that cannot be modified by a software adversary. It allows a collective attestation scheme to guarantee that all measurements are taken within the same time frame. Note that, a software adversary that is capable of tampering with the clock of a member D_i in $\mathcal{N}$ can trick V_a into believing that a measurement was taken at time $t_2 \neq t_1$, where t_1 is the actual time of measurement.

4.3.3.2 *Protocol Features*

Securing collective attestation also requires a set of features that should be satisfied by the design of the solution itself. These features are described below:

- *Protocol feature #1:* A mechanism that guarantees a correct collection of measurements. This can be done by sending all measurements from all members of $\mathcal{N}$ to the verifier V_a, which then checks whether the measurement of each member was once correctly integrated (see Section 4.2). A second approach for ensuring correct collection can be based on binding each attestation response to the attestation session during which it was generated. By allowing attest to generate only one response for each session which can be verified by only one member through interact, this approach enables the correct collection of measurements while maintaining distributed verification (see Section 4.1).

- *Protocol feature #2:* A mechanism that ensures time consistency of measurements. This requires allowing attest to bind the time of every measurement to its time of generation, securing time retrieval against software adversary through RROC, sharing view of time between V_a and $\mathcal{N}$ through clock synchronization, and allowing V_a to share and verify the expected time of measurement though init and verify respectively.

- *Protocol feature #3:* A mechanism that ensures the independence between the attestation procedures of each member of $\mathcal{N}$. This requires that interact does not allow the generation or verification of a measurement on a member D_i to be influenced by another member D_j, security of measurements are guaranteed on an end-to-end manner, and the key(s) of any member D_i are not known to other members of $\mathcal{N}$. Central verification used in Section 4.2 represents such a mechanism that ensures independence.

- *Protocol feature #4:* A mechanism for preventing an exploitable gap between attestation and service delivery. This can be done by enforcing the uninterrupted delivery of the service directly after executing an instance of the attestation protocol. In particular, every member D_i in $\mathcal{N}$ executes the attested piece of code directly after measuring it through attest as in SMART [52].

A collective attestation solution that implements the above protocol features over a network $\mathcal{N}$ whose members are equipped with the hardware features described in the previous section would satisfy the properties of Section 4.3.2 and hereby be secure against *Collective_Attestation_Forgery* according to Definition 4.7.

4.3.4 *Protocol Analysis*

The collective attestation solutions presented earlier in this chapter were not systematically analyzed in terms of security properties and provided features. This has made an accurate comparison between them particularly challenging. In the following we analyze the collective attestation solutions presented in Section 4.1 and Section 4.2. We first show how they comply to the definition of collective attestation presented in Definition 4.6 . Then we check whether they satisfy the identified properties that provide security against *Collective_Attestation_Forgery* defined in Definition 4.7.

Collective Attestation Solution (Section 4.1): This collective attestation solution is defined as follows:

- setup(·): Setup is done by O initiating every member with an authentication key pair, a public key certificate, a reference software configuration, a software configuration certificate, and the public key of O. When two members meet, they exchange reference software configuration, and use the authentication key pair to share a symmetric key.

- init(·): The protocol is initiated the verifier V_a choosing a random member D_1 of $\mathcal{N}$ to act the the only member of the attestation interface $\mathcal{I}_a$. V_a sends D_1 an attestation request formed of a fresh nonce.

- attest$(k, \cdot)$: When a member receives an attestation request from a neighbor, it measures its software state and authenticates the result with a MAC based on the symmetric key shared with that neighbor. D_1, that receives its attestation request from V_a, authenticates its measurement with a digital signature based on its secret key.

- interact$(k, \cdot, \cdot)$: When D_1 receives an attestation request from V_a, it generates and sends a new fresh nonce as an attestation request for every neighbor. These neighbors then generate new nonces and send them to their neighbors and so on, until an attestation request containing a different fresh nonce is received by every member of $\mathcal{N}$. On the other hand, after a device generates its attestation response through attest$(k, \cdot)$, it sends the response to the neighbor from which it received the request. On every member, responses from all neighbors are aggregated into

an intermediate response. D_1 aggregate intermediate responses into a global response.

- report$(k, \cdot)$: D_1 sends to V_a a global attestation response which is an aggregate all intermediate responses received from its neighbor along with its own software measurement.
- verify$(K, \cdot, \cdot)$: After verifying the digital signature on the received global attestation response using the public key of D_1, V_a can determine the number of $\mathcal{N}$'s members whose software has not been compromised.

Collective Attestation Solution (Section 4.2): In the following we only present the differences to the collective attestation solution in Section 4.1:

- setup$(\cdot)$: Members of $\mathcal{N}$ are initialized with an Optimistic Aggregate Signature (OAS) secret key (Section 4.2.2). V_a is setup with the aggregate public key of $\mathcal{N}$ and a secure token signed by O.
- init$(\cdot)$: V_a initiates collective attestation by sending D_1 a fresh nonce along with the secure token.
- attest$(k, \cdot)$: All software measurements are signed with an OAS signature based on the OAS secret key.
- interact$(k, \cdot, \cdot)$: When a member of $\mathcal{N}$ receives an attestation request, it verifies the signature of O and forwards the same nonce to its neighbors. Members aggregate measurements of their neighbors along with their own based on the OAS signature scheme.
- report$(k, \cdot)$: The global attestation response sent from D_1 to V_a is formed of an OAS signature of all members of $\mathcal{N}$.
- verify$(K, \cdot, \cdot)$: V_a uses the aggregate public key of $\mathcal{N}$ to verify the OAS signature and learn the IDs of $\mathcal{N}$'s members whose software is compromised.

Protocols Properties: We now check whether these solutions satisfy the properties that enable security under *Collective_Attestation_Forgery*:

- *Properties #1, #2, and #3:* The two solutions satisfy these properties. The solution in Section 4.1 requires interact to have access to the key(s) k. Further, interact is responsible for the generation of the random nonce which influences the measurement process. This solution assumes that all members of $\mathcal{N}$ are equipped with a lightweight security architecture that enables secure remote attestation. On the other hand, interact in Section 4.2 requires no access to k, and has no influence on the measurement process. In this solution, only members that should be attested are assumed to be equipped with a lightweight security architecture.

- *Property #4:* The two solutions satisfy this property. The solution in Section 4.1 is based on distributed verification and only allows counting the number of benign devices in $\mathcal{N}$. To avoid double counting, the solution uses a session ID to identify an attestation session. This ID is included in the attestation response. The lightweight security architecture guarantees that every member generates one attestation response directed to one single neighbor for every session ID. This is enabled by the fact that members of $\mathcal{N}$ share different symmetric keys with each of their neighbors. However, if the adversary could trick a member to share the same key with multiple neighbors, it can then allow the attestation response of this member to be overcounted. On the other hand, the verifier in Section 4.2 centrally verifies that every member reported its correct state only once. This is guaranteed by the properties of the OAS signature.

- *Property #5:* The two solutions do not satisfy this property. They allow software measurements on different members of $\mathcal{N}$ to be taken at different times. In fact, the difference between time of measurements on $\mathcal{N}$'s members is not upper bounded. Consequently, a mobile adversary may use the Hide & Seek strategy presented in Section 4.3.1 to evade detection by both solutions.

- *Property #6:* The solution in Section 4.2 satisfies this property while the solution in Section 4.1 does not. In Section 4.1 our solution relies on neighbors' verification and hop-by-hop aggregation based on symmetric keys, which implies that extracting the secrets of one member would allow other software compromised members to go undetected. On the other hand, the central verification strategy followed in Section 4.2, that gives interact no influence on the measurement process or access to the key(s), provides hardware independence between $\mathcal{N}$'s members.

- *Property #7:* The two solutions do not satisfy this property. They do not bind the attestation to any service delivery. As a consequence, a mobile adversary could compromise the software of members of $\mathcal{N}$ which leads to disrupting a specific service while not being detected by those solutions.

As can be seen the two solutions do not satisfy all the properties that provide security against *Collective_Attestation_Forgery*. Therefore, they are susceptible to various attacks through a stronger adversary. In Section 4.3.5 we present a generic collective attestation solution which completes the tradeoff between security/applicability and performance/assumptions by providing stronger security under *Collective_Attestation_Forgery*.

4.3.5 *Generic Solution*

The generic collective attestation solution $CA = (\mathsf{setup}, \mathsf{init}, \mathsf{attest}, \mathsf{interact}, \mathsf{report}, \mathsf{verify})$ is shown in Figure 4.22. The solution utilizes the Optimistic Aggregate Signature (OAS) scheme to provide secure collection and hardware independence, and synchronized Reliable Read-Only Clocks (RROC) for time consistency. The details of the solution are described in the following:

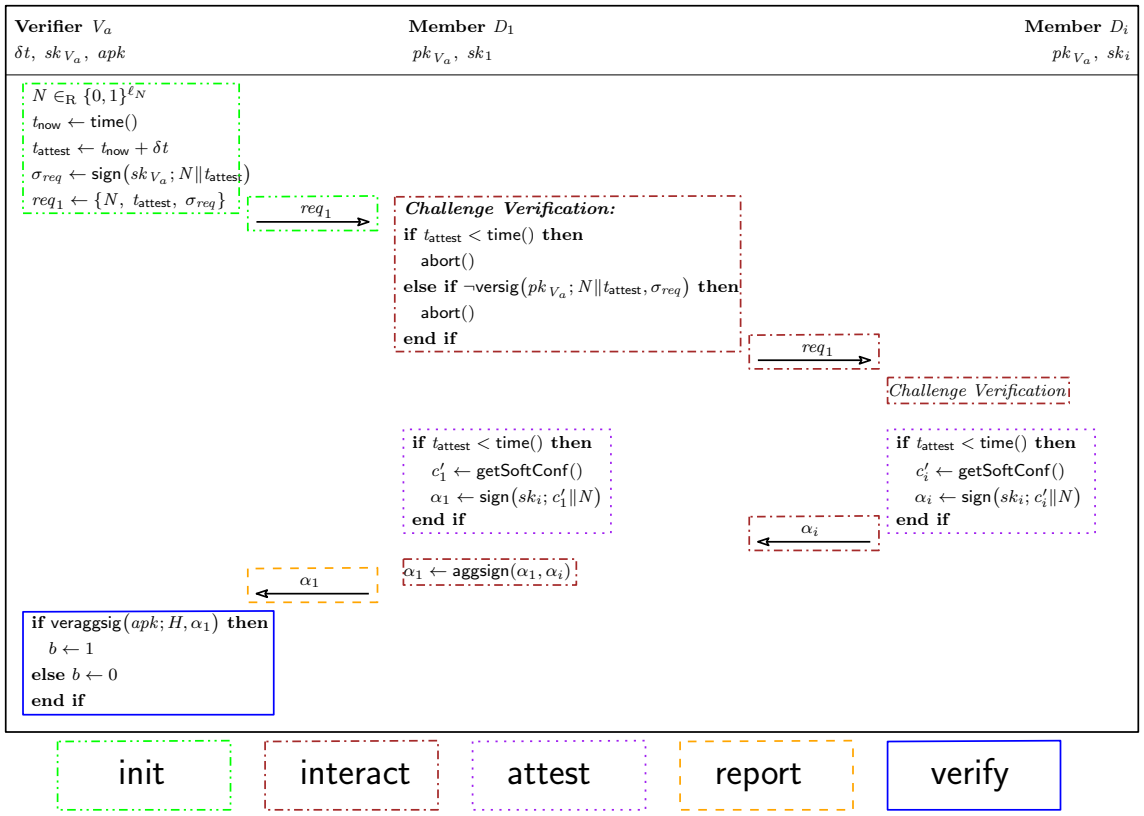


Figure 4.22: Protocol CA

- setup(·): Every member of $\mathcal{N}$ is initialized by O with a secure OAS key, and the public key of the verifier V_a. The verifier V_a is setup with the aggregate key of $\mathcal{N}$ and a signing key pair. V_a's setup can be done during initialization or using a dedicated protocol as in Section 4.2.

- init(·): V_a chooses a random member D_1 of $\mathcal{N}$ to act at the only member of the of the attestation interface $\mathcal{I}_a$ and sends it an attestation request $req_1 = \{N, t_{attest}, \sigma_{req}\}$. The request is formed of a fresh nonce N and the attestation time $t_{attest} > t_{now} + \delta t$ authenticated with a digital signature based on the secret key of V_a, where t_{now} is the time of generation of the attestation request and $\delta t$is the lower bound on the time needed for broadcasting the request to all members of $\mathcal{N}$.

- attest$(k, \cdot)$: At attestation time t_{attest}, every member of $\mathcal{N}$ that has received an attestation request from a neighbor (containing t_{attest}) measures its software state and authenticates the result with an OAS signature based on its secret OAS key.

- interact$(k, \cdot, \cdot)$: When D_1 receives an attestation request from V_a, it verifies the integrity of the signature of V_a, and the freshness of the attestation time t_{attest}. If the verification was successful, it forwards the request to its neighbors. The neighbors then verify and forward the request to their neighbors and so on, until the request is received and verified by every member of $\mathcal{N}$. On the other hand, after a device

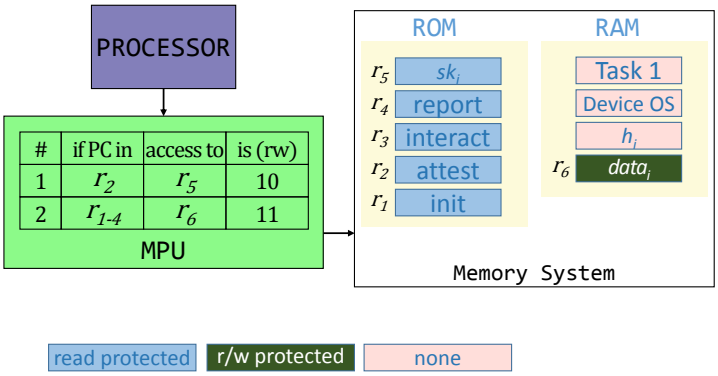


Figure 4.23: Implementation based on SMART [52]

generates its attestation response through attest$(k, \cdot)$, it sends the response to the neighbor from which it received the request. On every member, responses from all neighbors are aggregated into an intermediate response based on the OAS signature scheme. D_1 aggregates intermediate responses into a global response composed of one OAS signature.

- report$(k, \cdot)$: D_1 sends to V_a a global attestation response which is formed of an OAS signature of all members of $\mathcal{N}$.
- verify$(K, \cdot, \cdot)$: V_a uses the aggregate public key of $\mathcal{N}$ to verify the OAS signature and learn the IDs of $\mathcal{N}$'s members whose software is compromised.

In order to mitigate Time of Check Time of Use (TOCTOU) attacks, it is proposed that every member of $\mathcal{N}$ executes the attested code uninterrupted directly after executing attest.

4.3.6 *Implementation*

For the generic collective attestation solution we also present two instantiations on top of SMART [52] and TrustLite [84] security architectures (see Chapter 3). Recall that, these architectures provide their security guarantees based on minimal hardware features. The main features incorporated are a small amount of Read-Only Memory (ROM) and a simple Memory Protection Unit (MPU). In the following we present the two instantiations:

Implementation on SMART. For our implementation on SMART [52], we store in ROM of every member the program code responsible for executing the collective attestation on that member, i.e., init, attest, interact, and report. The ROM also stores for each member D_i the OAS secret key sk_i. Consequently, integrity of the measurement code and the OAS key are ensured via emutability of ROM. All protocol intermediate values are stored in rewritable memory of each member D_i. Note that these values have to be updated during D_i's lifetime. The implementation on SMART is shown in Figure 4.23 where we denote by RAM the rewritable memory region. SMART's MPU is configured such that it ensures secret data is only accessible to the unmodified part of code that

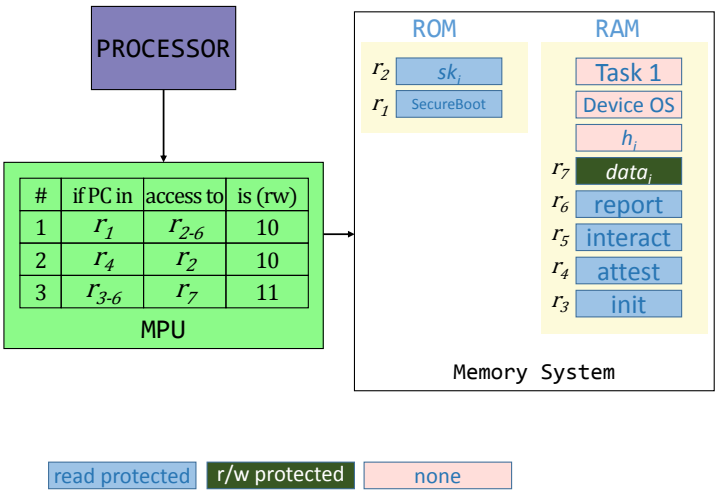


Figure 4.24: Implementation based on TrustLite [84]

requires access to this data. In particular, rule #1 ensures that only attest has read access to the OAS secret key sk_i, and rule #2 ensures that only init, attest, interact, and report have read and write access to their private data $data_i$.

Implementation on TrustLite. Our generic solution is also implemented as trustlets on TrustLite [84] (see Chapter 3). More precisely, we implemented the code responsible for executing each protocol/function, i.e., init, attest, interact, and report, as a single independent trustlet. Our implementation is shown in Figure 4.24. Trustlite ensures the software integrity of each of the protocols through the secure boot component SecureBoot on A_i. Further, as in SMART, the MPU of TrustLite was configured such that it ensures that secret data is only accessible to appropriate trustlets. In particular, rule #1 ensures that only SecureBoot has read access to the memory storing the program code of the generic solution, rule #2 ensures that only attest has read access to the OAS secret key sk_i, and rule #3 ensures that only init, attest, interact, and report have read and write access to their private data $data_i$.

4.3.7 *Performance Evaluation*

The overhead of the generic solution is compatible in terms of computational, communication, and energy costs with that of the solution presented in Section 4.2. However, its memory overhead is significantly reduced due to the elimination of monotonic counters (see Section 4.2.5). In this section we evaluate the runtime overhead of the generic solution based on simulation results for networks of up to $1,000,000$ devices. Our performance evaluation is based on the implementation from Section 4.3.6. Note that, we also assume here that the topology of the network does not change while the attestation protocol is executing.

We used the OMNeT++ [104] network simulator to assess the performance of the attestation protocol for very large networks of embedded devices. The attestation pro-

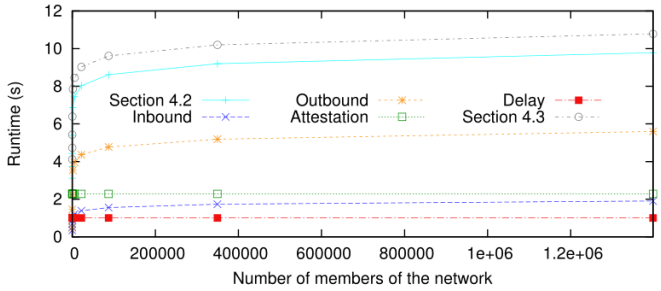


Figure 4.25: Runtime of the generic solution

tocol was again implemented on the application layer, where cryptographic operations were emulated with delays corresponding to real measurments of their execution time on TrustLite [84]. For our simulations, the average communication rate between devices was set to 250 Kbps, which corresponds to the defined bandwidth for ZigBee [135]. We considered three topologies for our evaluation: a star topology, a chain topology, and tree topologies with number of child nodes varying from 2 to 12. And, we simulated various network sizes, which ranged from 10 to $1,000,000$. However, since collective attestation is mostly relevant for tree topologies, we present our evaluation results only for a tree with 4 neighbors per device, and compare it to the results of the solutions presented in Section 4.2. Finally, we assume that each device has 50 KBytes of software to be attested. The results of our evaluation are shown in Figure 4.25.

Similar to the previous solutions, the runtime overhead of the generic solution is logarithmic in the size of the network. It can be as low as 11 s for attesting a network composed of $1,000,000$ members. Moreover, the generic solution shows a constant increase in runtime over the solution presented in Section 4.2. In order to understand the source of this increase, we divided the attestation protocol execution into three parts and measured the runtime required to execute each part. In particular, we distinguish between: (1) inbound execution which allows attestation requests to be broadcasted in the network $\mathcal{N}$ and verified by each member, (2) outbound execution through which attestation responses are aggregated in their way to the verifier, and (3) attestation which corresponds to the actual generation and authentication of the software measurement by each member of $\mathcal{N}$. In addition to these parts of the protocol execution, the generic solution imposes an additional delay that allows all members of $\mathcal{N}$ to generate their measurement at the same time, i.e., waiting till t_{attest} after receiving an attestation request. The runtime increase is mainly caused by this delay.

4.3.8 *Conclusion*

In this section we presented a systematic treatment of collective attestation that allows establishing such a security service on solid ground. This treatment enables analyzing and comparing the properties and features of the solutions presented earlier in this chapter. Further, we presented a generic collective attestation solution that satisfied all the security properties identified in this section. The generic solution completes the

tradeoff between security/applicability and performance/assumptions of collective attestation by providing stronger security guarantees at the cost of minimal additional requirements and runtime overhead.

4.4 RELATED WORK

In this section we present work from the literature that is directly related to the solutions presented in this chapter. A survey of existing attestation schemes is presented in Chapter 3.

Practicality of Collective Attestation. LISA [31] aims at investigating the applicability of collective attestation in real networks. It provides two practical instantiations of the collective attestation solution presented in Section 4.1. In particular, the main focus of LISA is the spanning tree construction. It comprises a synchronous instantiation which aggregates attestation reports as described in Section 4.1, and an asynchronous instantiation where attestation reports are directly forwarded without aggregation. The authors investigate the requirements and overhead of each of these two instantiations demonstrating the practicality of the solution described in Section 4.1.

Secure Data Aggregation. Secure data aggregation is a particularly popular research topic in the field of Wireless Sensor Networks (WSN). The goal of secure data aggregation schemes is to preserve the authenticity of data that is reported by sensor nodes while reducing the overall communication overhead in the network. Existing proposals are either based on new cryptographic techniques [72, 93, 34, 33, 156, 87, 106], incorporate a witness-based solution [51], or rely on trust relations [105]. Unfortunately, these approaches are based on computationally intensive public key cryptography [87], require a shared global key [93], or assume that all devices in the network are synchronized [51]. Finally, the majority of existing schemes impose high communication and computation overhead [51, 100, 36]. The solution we present in Section 4.1 aggregates attestation reports in a hop-by-hop manner as done by [92, 157], thus, avoiding the huge overhead of other existing approaches. However, we exploit a lightweight security architecture to provide end-to-end authenticity. Further, in Section 4.2 we leverage a novel aggregate signature scheme for aggregating attestation responses.

Random Sampling. As explained in Section 4.1.6, one can improve the efficiency of our attestation solution by leveraging random sampling, and verifying the integrity of a small subset of the whole network. McCune et al. use random sampling to detect when a sensor node does not receive a certain broadcast message [94]. In particular, in their solution only a randomly sampled subset of the sensor nodes acknowledge the receipt of the broadcast. The main intuition behind the security of such schemes is that it is not feasible for the adversary to find out in advance the particular sensor nodes that are expected to acknowledge the broadcast.

Sensor Networks (SN). A large amount of research has been done in the area of SN and WSN. This research mainly concerns topics like secure routing [73, 159], secure key management [154, 59], and secure broadcasting [7, 128]. To the best of our knowledge, our

solutions represent the first to provide scalable software integrity verification solutions in these areas or in any other area.

Distributed Signing The aggregate signature scheme of Boneh et al. [25] has already been used by Syta et al. in a distributed signing application [139]. However, the protocol presented by Syta et al. requires that all the signatures are generated on the same message. Moreover, the authors do not provide a detailed security proof of their protocol which is definitely more than just a formality.

4.5 CONCLUSION

In this chapter we investigated the problem of malware infestation and presented two collective attestation solutions that enable the detection of malware in large networks of embedded devices. The presented solutions are based on different primitives and assumptions, provide different security guarantees, have different cost, and are applicable in different scenarios. They provide a tradeoff between security and efficiency. Together the solutions enable software integrity checking for a wide range of applications. We further presented a systematic analysis of collective attestation that puts it on solid grounds and provides a guide for researchers working in this area. In Chapter 5, we investigate the problem of physical attacks on attestation, which is realistic in large-scale settings, and provide multiple collective attestation solutions that allow detection of physical attacks in various application scenarios.

5

DETECTION OF PHYSICAL ATTACKS

The collective attestation solutions presented in Chapter 4 adhere to the requirement of attestation in the single-prover setting by targeting malware attacks only. These solutions assume physical attacks on provers to be either not possible (Section 4.1) or very unlikely (Section 4.2). This assumption is reasonable in the single-prover setting, where devices can be unreachable or physically protected. It might also still hold for certain embedded networks where most of the devices are within secure premises and are far away from the reach of the adversary. However, emerging scenarios involve a large number of heterogeneous devices that are spread across a very large area. In such scenarios, it is not reasonable to assume that physical attacks are not possible. While some devices remain protected, others might become physically accessible to the adversary and within its grasp, e.g., public devices in building automation. In this chapter, we present two collective attestation solutions for large networks of embedded devices where physical attacks on devices are possible. The presented solutions allow the detection of both software and physical attacks in centralized and decentralized networks by leveraging attestation and absence detection. In Section 5.1, we present a solution for software and physical attack detection in centralized networks. The solution builds on top of collective attestation presented in Chapter 4 and extends it with periodic presence checks in order to enable detection of both attacks. Next, Section 5.2 presents a collective attestation solution for detecting software and physical attacks in autonomous networks. The solution exploits neighbors' attestation and presence checking to allow autonomous detection of these attacks. It further relies on a novel migration protocol in order to tolerate dynamic topologies.

Remark. The results presented in this chapter are due to the author of this work and the result of many intensive discussions and collaboration with Shaza Zeitouni (TU Darmstadt, Germany), Gene Tsudik (UCI, California), and Ahmad-Reza Sadeghi (TU Darmstadt, Germany). Parts of this chapter have been published in [76], [74], and [75].

5.1 DEVICE ATTESTATION RESILIENT TO PHYSICAL ATTACKS

It is not always enough to verify the software integrity of embedded networks in order to ensure their correct operation. Emerging embedded networks might involve devices that could be captured by the adversary and physically attacked. For example, devices placed in public regions in a building automation scenario, or those on the edge of the network in a smart factory or a perimeter monitoring scenario, are a natural target for physical attacks. Consequently, a collective attestation solution for such networks, that aims at verifying their safe and correct operation, should not only be secure under a stronger adversary that is capable of physical attacks, but also provide means for detecting those attacks.

The core observation we make in this chapter is that, physically attacking a device requires removing it from the network and turning it off for a non-negligible amount of time. In particular, the adversary must take the device apart and disassemble its components in order to extract its secrets [21]. Consequently, it is possible to detect a physical attack on a device by monitoring its presence in the network and detecting its absence. The solution presented in this section takes advantages of prior work on absence detection in Wireless Sensor Networks (WSN) [41, 42, 150, 67, 69, 70, 37, 147]. It targets centrally managed systems where multiple devices are physically accessible to the adversary, e.g., smart factories. Such networks are usually composed of a large number of heterogeneous devices. Thus, it is crucial to distribute the burden across all devices in the network in order to avoid performance bottlenecks.

Contribution. We investigate the security of large dynamic networks of heterogeneous embedded devices where physical attacks are possible. We define the threat model for these networks that allows physical attacks. And, we present the first collective attestation solution for centralized embedded networks that is capable of detecting both software and physical attacks. Our solution is based on collective attestation. It extends our collective attestation solutions presented in Chapter 4 with an absence detection protocol in order to provide security under a stronger adversary model and allow detection of both software and physical attacks. The presented solution allows detecting whether a physical attack has occurred, and can be extended to detect attacked devices, e.g., using majority voting. We present two instantiation of our solution based on digital signatures and Message Authentication Codes (MACs). Further, in order to demonstrate feasibility, we show two implementation of the solution on two recent security architectures for low-end embedded devices with different security features and functional capabilities: SMART [52] and TrustLite [84]. Finally, we present extensive performance evaluation based on our implementations, in addition to simulations of the solution in networks of up to $100,000$ devices demonstrating scalability. Our solution presents a first step towards secure detection of software and physical attacks in centralized embedded networks.

Outline. After providing a brief overview of our solution in Section 5.1.1, we present it details in Section 5.1.2, and describe our implementation in Section 5.1.3. Performance evaluation is then presented on Section 5.1.4. Security of the solution is examined in Section 5.1.5, and this section concludes in Section 5.1.6.

5.1.1 *Collective Attestation*

5.1.1.1 *Problem Description and System Model*

We consider a dynamic centrally managed network $\mathcal{N}$ that is formed of n interconnected heterogeneous devices D_i. The network operator is denoted by O. A central entity denoted by verifier V infrequently inspects $\mathcal{N}$ and assesses its trustworthiness. We do not assume the existence of a routing protocol. However, each device in $\mathcal{N}$ is at least capable of communicating to its direct neighbor [44, 68, 114, 115]. Since the mobility of devices can be involuntary, i.e., guided by ambient factors, neither the operator O nor the veri-

fier V are assumed to be aware of the network topology at any given time. Some of the devices in $\mathcal{N}$ are physically protected or inaccessible, while other devices are within the adversary's grasp and are subject to physical attacks. The goal of a collective attestation solution is to enable the verifier V to verify the software and hardware integrity of $\mathcal{N}$ in a secure and scalable manner. We consider a network to be trustworthy if all the devices are running software deployed and certified by O and none of them has been physically attacked.

5.1.1.2 *Requirements Analysis*

Objectives. Based on the model described above, a collective attestation solution for large networks of embedded devices should provide, in addition to the five properties described in Section 4.1.1, the following properties:

- *Property #1:* Allow verifying the collective integrity of the network $\mathcal{N}$.
- *Property #2:* Detect and identify devices whose software has been compromised.
- *Property #3:* Be more efficient than individual attestation of every device in $\mathcal{N}$.
- *Property #4:* Allow detection of devices whose hardware has been physically attacked.
- *Property #5:* Report and allow verification of the network topology.

The satisfaction of property #1, property #2, and property #3 pertain to main security objective of collective attestation presented in Chapter 4. The solution proposed in Section 4.1 aims at satisfying these properties in a software-only threat model. Further, the solution presented in Section 4.2 attempt to satisfy these properties even when a limited number of devices can be physically attacked. Neither of these solutions satisfy property #4 and property #5. While property #4 is crucial in the presence of a powerful adversary that is capable of capturing and physically attacking the hardware of a large number of devices in $\mathcal{N}$, property #5 allows verifying the integrity of $\mathcal{N}$'s topology, which might be considered in certain scenarios an important part of $\mathcal{N}$'s integrity.

Adversary Model. As in Chapter 4 we assume that the adversary $\mathcal{A}$ has complete control over the communication channel, i.e., it can eavesdrop on, modify, drop, or replay any message exchanged between all devices in $\mathcal{N}$ and between any device D_i and the verifier V. We assume that the operator O and the verifier V are trusted. Further, we consider three kinds of adversaries targeting the devices in $\mathcal{N}$:

- Software-only adversary $\mathcal{A}_{v,0}$: This adversary represents the standard adversary in all prior attestation protocols including collective attestation presented in Chapter 4. $\mathcal{A}_{v,0}$ is capable of compromising the software of $v \leqslant n$ devices through remote software attacks.
- Hardware-only adversary $\mathcal{A}_{0,w}$: This adversary has physical access, and is hereby capable of capturing and physically attacking the hardware of $w < n$ devices.

- Hybrid adversary $\mathcal{A}_{v,w}$: This adversary can compromise the software of v devices and physically attack the hardware of w devices.

Let the inter-attestation gap t_{att} denote the upper bound on the time between two successive inspections of $\mathcal{N}$ by V, which indeed executes collective attestation to assess $\mathcal{N}$'s trustworthiness. The parameter v (resp. w) refer to the upper bound on the number of devices whose software (resp. hardware) can be attacked by $\mathcal{A}$ within t_{att}. Moreover, we denote by t_{phy} the non-negligible amount of time that is required by $\mathcal{A}$ to physically attack the hardware of one device. We place some constraints on the capabilities of $\mathcal{A}$. In particular, the following are out of scope:

1. Omnipotence: We claim that the strongest adversary that can be mitigated is $\mathcal{A}_{n,n-1}$, which compromises the software of all devices and physically attacks the hardware of all but one device. Therefore, an almighty adversary $\mathcal{A}_{0,n}$ that can physically attack the hardware of all devices within t_{att} is considered out of scope.
2. Non-invasive physical attacks: We consider physical attacks that do not require turning off the device for a non-negligible amount of time to be out of scope. Examples of such attacks include hardware side-channel attacks that aim at extracting a device's secrets during its normal functioning. These attacks represent an orthogonal problem.
3. Denial of Service (DoS): We assume a stealthy adversary that aims at attacking the largest number of devices while remaining undetected. Therefore, we consider DoS attacks to be out of scope.

Device Requirements. Through the design of our solution we satisfy property #3 and property #5. However, in order to satisfy property #1 and property #2 as well as the five properties described in Section 4.1.1, it should be possible to securely attest each device in $\mathcal{N}$. Consequently, every device D_i should satisfy the requirements for secure remote attestation as discussed in Section 4.1.1. Further, satisfying property #4 requires means of sharing time between devices, which should also be secure against software attacks when these attack are considered within the adversary model. We now list all the requirements that should be satisfied by each D_i in $\mathcal{N}$:

- Reliable Clocks: Devices in $\mathcal{N}$ should be equipped with reliable clocks that are synchronized with the clock of the verifier V. We denote by δ_t the upper bound on the clock skew between any two devices in $\mathcal{N}$ and between $\mathcal{N}$ and V.
- Reliable Read-Only Clocks (RROC): A RROC is a reliable clock that cannot be modified by any software that is residing on the device, i.e., it is non-malleable.
- Lightweight Security Architecture: Each device is equipped with a lightweight security architecture that enables secure remote attestation, e.g., SMART [52] and TrustLite [84] (see Chapter 3).

If the first requirement is not achieved, V might not detect whether a hardware-only adversary has physically attacked the hardware of a device D_i. If the second requirement

is not achieved, V might not detect whether a hybrid adversary has compromised the software or physically attacked the hardware of a device D_j. And, if the third property is not achieved V might not detect whether a software-only adversary has compromised the software of a device D_k.

Assumptions. As mentioned in the system model, devices in $\mathcal{N}$ can be heterogeneous in terms of hardware and/or software. However, each device should satisfy the requirements that allow secure detection of attacks under the specified adversary model. In particular software-only adversary requires every device to have a lightweight security architecture, hardware-only adversary requires reliable clocks in addition to the lightweight security architecture, and hybrid adversary requires a synchronized RROC in addition to the security architecture. We denote the maximum clock skew between any two devices by δ_t. Devices in $\mathcal{N}$ can communicate with each other, i.e., any two devices D_i and D_j always have a communication path between them. The time required to transmit a message between two devices is upper bounded by t_{tr}. Further, while the network topology is assumed dynamic, during the execution of the attestation protocol the network is assumed connected and its topology should remain static. Finally, cryptographic primitives are assumed to be secure along with their implementations.

Protocol Overview. The core idea is that during the time when the network $\mathcal{N}$ is unattended by V, devices in $\mathcal{N}$ periodically monitor each others presence. Eventually, when the network is attended, V performs collective attestation of $\mathcal{N}$ as described in Chapter 4 and collects presence reports from all devices. The solution proposed in this section consists of two protocols: beat and attest. beat allows every device to monitor the presence of all other $n-1$ devices in $\mathcal{N}$. In particular, each device periodically broadcasts a heartbeat to all other devices in the network through its immediate neighbors, thus proving its presence in $\mathcal{N}$ during a specific period of time. Devices verify and log all heartbeats received from all other devices for all time periods. attest allows V to collectively attest all devices in $\mathcal{N}$. Alongside V collects all heartbeats logged by all devices in $\mathcal{N}$. Through the collected logs, V can detect all devices that were not present in $\mathcal{N}$ for an extended period of time. These devices are assumed to be physically attacked.

5.1.2 *Protocol Description*

We now describe in details the protocols involved in our solution. In order to motivate different device requirements, we present three different versions of our solution that are secure against the three different kinds of adversaries described in Section 5.1.1.

5.1.2.1 *Mitigation of Hardware-Only Adversary*

Lets assume a hardware-only adversary $\mathcal{A}_{0,n-1}$ as described in Section 5.1.1, i.e., an adversary that can physically attack all but one device in $\mathcal{N}$ within the inter-attestation gap t_{att}, while being incapable of compromising the software of that remaining device. As mention earlier, in order to mitigate this adversary each device should be equipped with a lightweight security architecture and a reliable clock.

Intuition. The core idea of our solution is that if we know the minimum time t_{phy} that is required by $\mathcal{A}_{0,n-1}$ to physically attack a device D_i, then running an absence checking protocol at periods t_{hb} that are shorter than t_{phy} would allow the detection of physical attack on D_i. Consequently, if these two conditions hold:

- All heartbeats generated by a devices whose hardware is not physically attacked by $\mathcal{A}$ are unforgeable and uniquely bound to their generation time.
- All devices generate heartbeats in and timely manner (i.e., every $t_{hb} < t_{phy}$) and log all heartbeats received from all other devices.

Then the following assertion can be safely made

> **Assertion 1:** [**Alert**] Assume that w devices are captured within a given inter-attestation gap t_{att}, where $0 < w < n$. Then the heartbeat logs of at least one device D_i, whose hardware has not been physically attacked, will be missing at least one heartbeat of one other device D_j that corresponds to one heartbeat period.

If a device D_j is physically attacked, then D_j should be absent for at least t_{phy}. Since $t_{phy} > t_{hb}$, the log of D_i will lack one heartbeat of D_j which corresponds to the heartbeat generation time during which it was switched off due to the physical attack. Indeed, the physical attack on D_j will allow $\mathcal{A}$ to extract its secrets and be able to re-generate the missing heartbeat. However, it would then be too late as the absence of D_j has been already recorded by D_i.

It is important to note that the inverse of this assertion does not always hold. In particular, if a heartbeat of D_j is missing from D_i's logs this does not imply that the hardware of D_j has been physically attacked. A log inconsistency could be caused by other factors such as network problems or device failures. Therefore, *false positives* are possible while *false negatives* are not.

Moreover, the following assertion can be also made:

> **Assertion 2:** [**Normalcy:**] After the execution of collective attestation, if all the reported logs of every device in $\mathcal{N}$ contain all the heartbeats for all devices for every t_{hb} period, then none of the devices was absent for a period longer than t_{hb}.

Since $t_{phy} > t_{hb}$ the second assertion implies that $\mathcal{A}$ has not physically attacked the hardware of any device in $\mathcal{N}$. Consider one device D_i whose hardware has not been physically attacked. If the log of D_i contains a heartbeat of every other device for every period since the last time the network was attended, then none of the devices has been absent for longer than t_{hb}. However, D_i is not known to V. Since all logs of all devices match and at least one device is not physically attacked, then none of the devices has been absent and hereby physically attacked.

Figure 5.1 : protocol beat (as viewed by D_i)

timeOut($timer_{hb}$)

startTimer ($timer_{t_{tol}}, t_{tol}$)

$t_i = t_{start} = time()$, $hb_i = \{p, t_i, id_i\}$

$\sigma_i = sign(sk_i; hb_i)$

Set $log_i = hb_i$, startTimer ($timer_{hb}, t_{hb}$)

$D_i \xrightarrow{hb_i,\ \sigma_i}$ all neighbors

while ($\neg$ timeOut ($timer_{t_{tol}}$) $\wedge$ $sizeof(log_i) < n$) **do**

  $D_i \xleftarrow{hb_j,\ \sigma_j}$ neighbor (D_j)

  if ($accept(t_{start}, t_j) \wedge hb_j \notin log_i$) **then**

    if versig(pk_j, hb_j, σ_j) **then**

      $Append(\{hb_j, \sigma_j\},\ log_i)$

      $D_i \xrightarrow{hb_j,\ \sigma_j}$ all neighbors

    else

      Discard hb_j

    end

  else

    Discard hb_j

  end

end

$p = p + 1$

Heartbeat. Periodically, i.e., every t_{hb} time interval, each device D_i generates a heartbeat and broadcasts it to its neighboring devices. Every heartbeat has a timestamp indicating its generation time and is signed by D_i based on its secret signing key sk_i. On the other hand, when a device D_i receives a heartbeat from its neighboring device D_j it verifies its freshness and authenticity by verifying the timestamp and the signature respectively. If the verification of the heartbeat was successful, D_i stores it in its log and forwards it to all its neighbors. D_i terminates the heartbeat protocol when it has receive and stored heartbeats from every other device in $\mathcal{N}$. The termination of the protocol can also be based on a timeout denoted by the tolerance interval t_{tol}. If D_i does not receive a heartbeat from some peer device D_j within t_{tol} after the start of the protocol execution, D_i concludes that D_j has been absent. This protocol is denoted by beat and is described in details in Figure 5.1.

Let id_i denote the unique ID of a device D_i, and let t_{tr} denote by the upper bound on the time required to transmit a heartbeat between any two devices in $\mathcal{N}$. Since the upper bound t_{phy} on the time required by $\mathcal{A}$ to physically attack the hardware of one device is considerably larger than t_{tr}, no tight upper bound on network delay is required. Note that, beat can be either initiated based on a timeout, i.e., expiry of a timer indicating that t_{hb} has passed since beat was last executed, or upon receipt of a heartbeat from one of the neighbors. For the sake of clarity, Figure 5.1 shows the protocol starting based on a timeout. We assume that all the heartbeats that are received before step 0 in Figure 5.1 are buffered.

Figure 5.2 : protocol collect (as viewed by V)

for $0 < i \leqslant n$ **do**
  $t = time(),\ N \in_R \{0,1\}^{\ell_N}$,
  $\sigma_V = \mathsf{sign}(sk_V; t \| N)\}$
  $V \xrightarrow{Ch=\{t,N,\sigma_V\}} D_i$
  $V \xleftarrow{resp_i=\{LOG_i,\sigma_i\}} D_i$
  if $\left(\mathsf{versig}(pk_i; LOG_i \| Ch, \sigma_i)\right)$ **then**
    Store LOG_i
  else
    Discard $resp_i$
  end
end

Attestation. As mentioned earlier, $\mathcal{N}$ is infrequently attended by the verifier V which inspects it and assesses its trustworthiness. When present, V collects all the heartbeat logs from all device in $\mathcal{N}$. V initiates collection by generating a collection request $Ch = N, t, \sigma_V$ and sending it to a device D_i in $\mathcal{N}$. N denotes a random fresh nonce, t a new timestamp, and σ_V a digital signature generated by V over N and t using its secret key sk_V. Note that, V can be remote or has physical proximity to D_i.

Figure 5.3 : protocol collect (as viewed by D_i)

$D_i \xleftarrow{Ch=\{t,N,\sigma_V\}} V$

if *If* $\left(\mathsf{versig}(pk_V; t\|N, \sigma_V)\right)$ **then**

  $\sigma_i = \mathsf{sign}(sk_i; \mathsf{LOG}_i\|Ch)$

  $D_i \xrightarrow{resp_i=\{LOG_i,\sigma_i\}} V$

else

  Discard Ch

end

When D_i receives the request Ch, it verifies its freshness and authenticity by verifying the timestamp and the signature. If the verification of the request was successful, D_i uses its secret key sk_i to sign the set $\mathsf{LOG}_i = \{\log_i, \forall p\}$ of all logs collected since the last collection instance along with the received challenge Ch. The signature and the log are then sent to V which verifies the authenticity and freshness of the signature and stores the received logs. Based on all logs collected from all devices, V can identify the devices that were absent during at least one heartbeat execution since it last attended $\mathcal{N}$. According to the second assertion, if the logs of all devices contain all heartbeats of all other devices, V concludes that $\mathcal{A}$ did not physically attack the hardware of any of the devices. This protocol is denoted by collect and is described in details in Figure 5.2, and 5.3.

Efficiency Improvements. For the sake of clarity, the protocols described above involved several costly operations and impractical assumptions. In the following we present several intuitive protocol modifications that provide substantial simplifications and cost reductions.

- Communication mode: The collection protocol described in Figure 5.2 and 5.3 assumes that the verifier V collects the logs directly from each of the devices in $\mathcal{N}$. While this idealized communication setting is plausible, it is in fact unrealistic. However, since both the collection request (Ch) sent by V to any device D_i and the response $resp_i$ of D_i are authenticated, an authentic channel is created between V and D_i. As a consequence, the security of the collection protocol remains intact regardless of the communication setting, i.e., even if the messages between V and D_i went through one or more intermediate communication hops.

- Public key cryptography: The two protocols described above are based on digital signatures, which is also mainly done to simplify presentation. Replacing digital signature with MACs in the beat protocol is in fact straightforward. It only requires that every device D_i in $\mathcal{N}$ shares a symmetric key with each of its neighboring devices. Similarly, having that each device shares a unique symmetric key with V, the digital signatures in the collect protocol can be simply replaced with MACs. Note that, under our hardware-only adversary $\mathcal{A}_{0,n-1}$ that is not capable of physically

attacking the hardware of *all* devices, MACs are as secure as digital signatures. In particular, if the hardware of at least one devices D_i is not physically attacked, the heartbeat log of D_i would lack at least one heartbeat of one absent device. The drawback of replacing digital signatures with MACs is that MACs require either a key pre-distribution or a key establishment protocol to enable sharing symmetric keys between neighboring devices.

- Heartbeat logs: The collect protocol description in Figure 5.2 and 5.3 shows that each device D_i should collect and send to V logs LOG_i of all other devices which are accumulated over and extended period of time through multiple executions of the beat protocol. While this is also done for the ease of presentation, it is sufficient for D_i to send V during collect a list of absent or present devices. In fact, D_i could simply send V a single bit signifying whether the hardware of at least one device in $\mathcal{N}$ is physically attacked.

In Section 5.1.2.3 we show a version of our protocol that considers all these improvements.

5.1.2.2 *Mitigation of Software-Only Adversary*

Lets now assume a software-only adversary $\mathcal{A}_{n,0}$ as described in Section 5.1.1, which in contrast to $\mathcal{A}_{0,n-1}$ can compromise the software of all devices in $\mathcal{N}$ but is incapably of physically attacking the hardware of a single device.

As already discussed, in order to mitigate this adversary each device in $\mathcal{N}$ should be equipped with a lightweight security architecture that enables secure remote attestation. In particular, on every device D_i the secrecy of the cryptographic keys and the integrity of the measurement and reporting mechanisms should be protected against software attacks. This can be achieved using minimal features such as a Read-Only Memory (ROM) for preserving the integrity of the measurement and reporting code, and a simple Memory Protection Unit (MPU) for preserving the secrecy of the involved cryptographic keys [52, 16, 84] (see Chapter 3 for more details).

If this requirement is satisfied, collective attestation presented in Chapter 4 can be used to detect all devices whose software has been compromised. In particular, V picks a random device D_1 as initiator and sends it an attestation request containing a random nonce N. The request is then flooded across the network and a spanning tree rooted at initiator is formed. Next, starting at leaf nodes in the tree, each device creates an attestation report and sends it to its parent. The reports are aggregated and propagate the tree in reverse en route to the verifier V which receives a single aggregated report. Upon verifying the report, V can determine the number (Section 4.1) or the IDs of the compromised devices (Section 4.2).

5.1.2.3 *Mitigation of Hybrid Adversary*

We know assume a hybrid adversary $\mathcal{A}_{n,n-1}$ as described in Section 5.1.1, i.e., an adversary that can, within the inter-attestation gap t_{att}, compromise the software of all devices and physically attack the hardware of all but one device. Let D_b be the single device

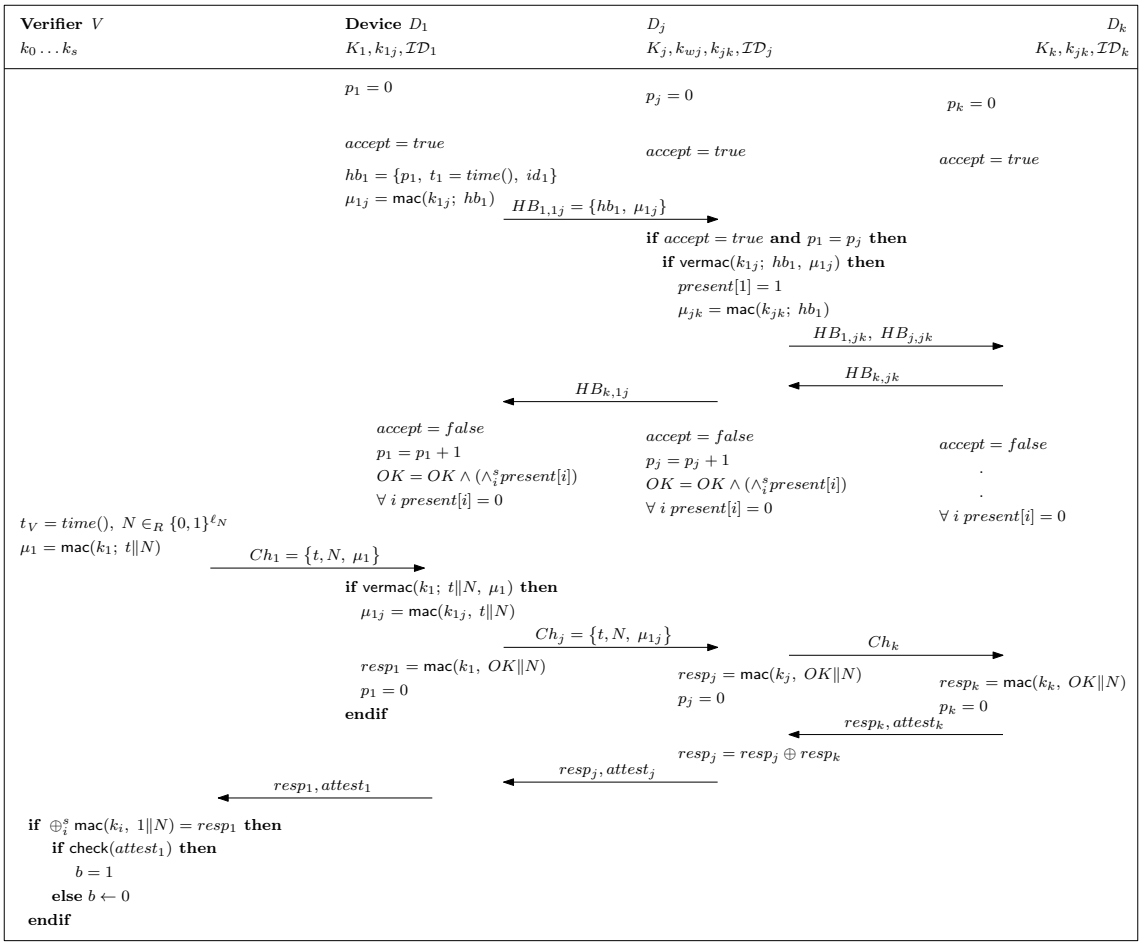


Figure 5.4: Protocol attest

whose hardware has not been physically attacked by $\mathcal{A}_{n,n-1}$. An obvious approach for mitigating $\mathcal{A}_{n,n-1}$ is to combine collective attestation from Chapter 4 with absence detection described in Section 5.1.2.2, thus detecting both software compromise and physical attacks.

In doing so, we also combine the respective assumptions and requirements needed to mitigate the two respective (non-hybrid) adversary types. In particular, the combined solution requires each device to be equipped with a ROM for preserving the integrity of the code used for both attestation and absence detection, a simple MPU that preserves the secrecy of the private protocol data including secret keys, and a reliable clock synchronized with the clock of V. In the following we describe the combined protocol in details. The protocol involves no public key operations and is based solely on symmetric cryptography. We assume that every device D_i shares a symmetric key k_{ij} with every neighboring device D_j. D_i also shares a key k_i with the verifier V.

Collective Attestation. Figure 5.4 shows the details of the combined protocol denoted by attest. Every device (e.g., D_1 in the figure) periodically, i.e., every t_{hb} interval, generates a heartbeat $hb_1 = \{p, t_1, id_1\}$, where p_1 is a monotonic counter identifying the

current heartbeat interval, t_1 is a timestamp indicating the time of generation of hb_1, and id_1 is the ID of D_1 that uniquely identifies it. For every neighbor D_j, the heartbeat is authenticated with a MAC μ_{1j} based on the key k_{1j} shared with D_j and then sent to that neighbor. Upon receiving an authenticated heartbeat from D_1, D_j verifies its authenticity and freshness by (1) verifying the MAC μ_{1j} and (2) ensuring that the heartbeat corresponds to the correct heartbeat interval p_j and was received within the tolerance interval, i.e., $accept = true$.

If the verification of the heartbeat was successful, D_j notes the presence of D_1 for the current heartbeat interval indicated by p_j ($present[1] = 1$). D_j then creates a new MAC over the heartbeat of D_1 and sends it along with hb_1 to all its neighbors. Finally, upon the expiry of a timeout ($t = 2 \cdot \delta_t + t_{tr}$), every device D_j reviews its list $present[]$ of present devices. If the list indicates that at least one device was not present during the current heartbeat interval, D_j notes this absence by setting the bit OK to 0.

When the network $\mathcal{N}$ is attended, V executes a collective attestation and collection protocol with $\mathcal{N}$. V chooses an arbitrary initiator device D_I and sends it an attestation request $Ch_1 = \{t, N, \mu_1\}$ containing a fresh timestamp t, a fresh random nonce N, and a MAC μ_1 over t and N based on the symmetric key k_1 shared with D_I. Upon receiving the request Ch_1, D_I verifies the MAC μ_1 and creates, for every neighbor D_j, a new challenge $Ch_j = \{t, N, \mu_{1j}\}$ that is authenticated with a MAC μ_{1j} based on the key k_{1j} shared with D_j. D_I then sends Ch_j to D_j. The neighbors repeat this process securely propagating the request to every device in $\mathcal{N}$. This procedure forms a spanning tree rooted at D_I.

Next, starting at leaf nodes, every device D_k authenticates the bit OK, which indicates whether any of the devices in $\mathcal{N}$ was absent for at least one heartbeat interval, along with the received nonce N based on the key k_k shared with V, thus generating $resp_k$. Moreover, D_k generates an attestation response $attest_k$ based on collective attestation described in Chapter 4. D_k then sends its response including $resp_k$ and $attest_k$ to its parent node in the spanning tree. Parents accumulate responses received from child nodes by XORing every $resp$ with their own and aggregating every $attest$ according to the description of the collective attestation solution. Consequently, the results propagate the spanning in reverse en route to V, which receives and aggregated response $resp_1$ formed of XORed MACs and a collective attestation report $attest_1$. In order to verify the response, V re-generates and XORs all the MACs. It then compares the outcome to the received $resp_1$. If the two match, V concludes that none of the devices in $\mathcal{N}$ was physically attacked. V can then verify the collective attestation response and determine the number (see Section 4.1) or IDs (see Section 4.2) of devices whose software has been compromised.

In order to allow devices to attest each other and enable software update, every device in $\mathcal{N}$ is initialized with a software configuration certificate. A software configuration certificate indicates the software configuration of a device and is signed based on V's public key. When a device first joins the network, it broadcasts this certificate to all its neighboring devices. Every neighbor then verifies the certificate and stores the software configuration for future attestation. On the other hand, avoiding double counting of devices may require a session identifier to be sent along with the attestation request.

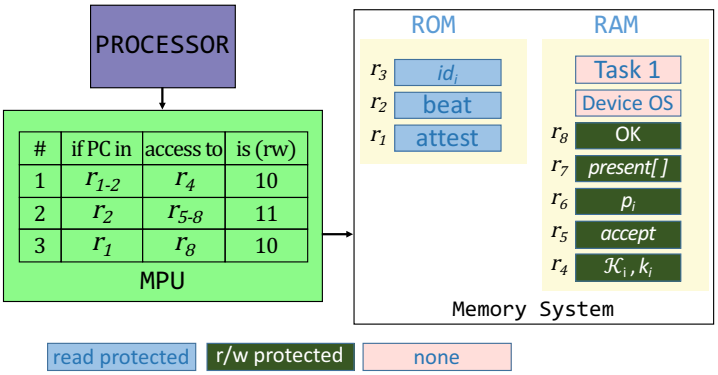


Figure 5.5: Implementation based on SMART [52]

This identifier should also be included in all attestation responses. For the ease of clarity, these details were omitted from the protocol description. However, they can be found in Chapter 4.

Discussion. Even though the protocol described above might look secure against a hybrid adversary, it is in fact not. The reason for this is that the reliable clock assumption is not enough to mitigate $\mathcal{A}_{n,n-1}$.

Assume that $\mathcal{A}$ physically attacked the hardware of a device D_i during the heartbeat interval p. Therefore, $\mathcal{A}$ will have access to all secret keys of D_i. If $\mathcal{A}$ is capable of compromising the software of all devices in $\mathcal{N}$, it can roll back their clocks in order to expand the tolerance interval during which a heartbeat is accepted. As a consequence, $\mathcal{A}$ can then generate D_i's missing heartbeat, which will be accepted by all compromised devices, since it appears to be received in the correct time. Numerous attacks can be executed on that protocol, common to all of them is that $\mathcal{A}$ can tamper with the clock of devices through remote software attacks.

As a consequence, mitigating a hybrid adversary requires that every device in $\mathcal{N}$ is equipped with a Reliable Read-Only Clock (RROC) as described in Section 5.1.1. We claim that RROC is both necessary and sufficient in the presence of such a strong adversary. Note that, by RROC we do not mean a secure clock that is physically protected or tamper evident. We simply require a clock that is not modifiable by software means. Such a requirement can be realized through the cmmercially available Real-Time Clocks (RTC) [4].

5.1.3 *Implementation*

We now present our implementation of the combined solution presented in Section 5.1.2.3 on top of two lightweight security architectures for low-end embedded systems: SMART [52] and TrustLite [84] (see Chapter 3). As already discussed, these two architecture we chose provide strong security guarantees for remote attestation based on minimal features in hardware, i.e., a small amount of Read-Only Memory (ROM) and a simple Memory Protection Unit (MPU).

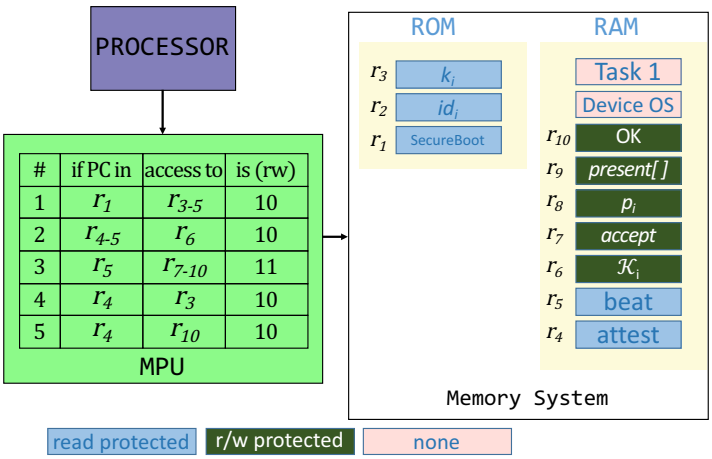


Figure 5.6: Implementation based on TrustLite [84]

Implementation on SMART. Our implementation on SMART [52] requires the same modifications to SMART's architecture presented in Section 4.1.3, i.e., the MPU is extended to control access to a small amount of rewritable memory. We require secure rewritable memory to store private data of both collective attestation, e.g., session identifiers, and absence detection, e.g., list of present devices. For this implementation we stored in ROM of each device D_i the program code, i.e., the code responsible for the execution of attest and beat, and ID id_i of D_i. Consequently, the integrity of the program code and the ID is ensured by the emutability of ROM. Moreover, we store the keys $\mathcal{K}_i = \{k_{ij}, \cdots, k_{ik}\}, k_i$ shared with neighboring devices and with the verifier V in the rewritable memory of every device D_i since this list could be updated during the lifetime of D_i. Our implementation on SMART is shown in Figure 5.5 where we denote rewritable memory by RAM. We configured SMART's MPU to ensures that private data is only accessible to unmodified code in ROM that needs to access this data. For example, rule #1 ensures that only attest and beat have read access to the keys $\mathcal{K}_i, k_i$, and rules #2 and #3 ensure that beat has read and write access to its private data, one of which, i.e., OK is read accessible to attest.

Implementation on TrustLite. This solution was also implemented as trustlets on the TrustLite security architecture [84] (see Chapter 3). More precisely, we implemented each of the protocols attest and beat as a single independent trustlet on device D_i. Our implementation is shown in Figure 4.5. In TrustLite, the integrity of each of these protocols is ensured via the secure boot component SecureBoot on D_i. Moreover, similar to SMART, the MPU of TrustLite ensures that secret data of D_i is only accessible to the appropriate trustlets. For example, rule #1 ensures that only SecureBoot has exclusive read access to the memory storing the program code attest and beat. rule #2 ensures that only attest and beat have read access to the keys in $\mathcal{K}_i$, rules #3 and #5 ensure that beat has read and write access to its private data, one of which, i.e., OK is read accessible to attest, and rules #4 ensures that attest also has read access to the key k_i shared with V.

5.1.4 *Performance Evaluation*

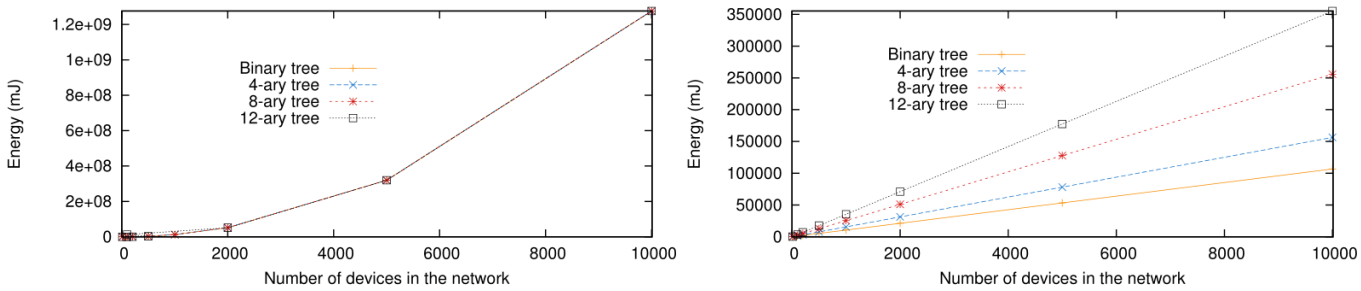
We assess performance of our solution in terms of computational, communication, memory, and energy costs. Further, we present simulation results for networks of up to 100,000 devices. The performance evaluation we present in this section is based on our implementation described in Section 4.1.3. It assumes that the topology of the network is static during the execution of the protocols.

Computation Cost. Cryptographic operations, such as generation of a MAC or a digital signature constitute the major part of the computation cost. Let g_i denote the number of neighbors of every device D_i, and $h_i \leqslant g_i - 1$ be the upper bound on the number of children of D_i in the spanning tree. In the instantiation based on digital signatures, each device D_i creates one digital signature and verifies n while executing beat. During collect, D_i creates one signature and verifies $h_i - 1$. Further, in the instantiation based on MACs, D_i creates n MACs and verifies n while executing beat. During attest, D_i creates $g_i + 1$ MACs and verifies $h_i + 1$.

Communication Cost. We used HMAC based on SHA-1 as our MAC implementation, and ECDSA as our digital signature scheme, i.e., $\ell_{\mathsf{mac}} = 160$ and $\ell_{\mathsf{sign}} = 320$. We further used a 64 bit timestamp and chose $\ell_N = 160$ and $\ell_p = 64$. As a consequence, nonces, and MACs, are 20 Bytes each. The variables t, p and *id* are 8 Bytes each. And, digital signatures are 40 Bytes each. In the instantiation based on digital signatures, each device D_i has a communication overhead, which is upper bounded by receiving $72g_i$ Bytes and sending $72g_i$ Bytes in beat, and receiving $68g_i + (40 + 72n \cdot \#hb \cdot x)h_i$ Bytes and sending $108 + 72n \cdot \#hb \cdot x$ Bytes in collect. $\#hb$ represents the number of heartbeat intervals within the inter-attestation gap t_{att}, and $0 < x \leqslant n$ denotes the number of logs in the response which depends on D_i position in the spanning tree. Further, in the instantiation based on MACs, D_i has a communication overhead, which is upper bounded by receiving $44n$ Bytes and sending $(24 + 20g_i)n$ Bytes in beat, and receiving $48g_i + 20h_i$ Bytes and sending $28 + 20g_i$ Bytes in attest.

Memory Cost. In the instantiation based on digital signatures, each device D_i in $\mathcal{N}$ should store: (1) an authentication key pair (sk_i, pk_i) and the corresponding identity certificate $\mathrm{cert}(pk_i)$; (2) the public key pk_V of the verifier V; and (3) and the log LOG_i of all heartbeats collected within t_{att}. The memory cost of this instantiation is around $100 + 72n \cdot \#hb$ Bytes. Further, in the instantiation based on MACs, D_i should store: (1) keys $k_{ij}, \cdots, k_{ik}, k_i$ shared with neighboring devices and with V; (2) a bit string present[] indicating which devices where present during the current heartbeat interval; (3) a monotonic counter p that identifies the current heartbeat interval; (4) a flag accept that determines whether we are with the tolerance interval; and (5) a flag OK indicating whether at least one device was absent for at least one heartbeat interval. The memory cost of this instantiation is around $226 + 160g + n$ bits.

Energy Cost. We estimated the energy consumption of the solution presented in Section 5.1.2.3 based on the energy costs of communication and cryptographic operations reported for two sensor nodes: MICAz and TelosB [47], which belong to the same class



(a) Instantiation based on signatures

(b) Instantiation based on MACs

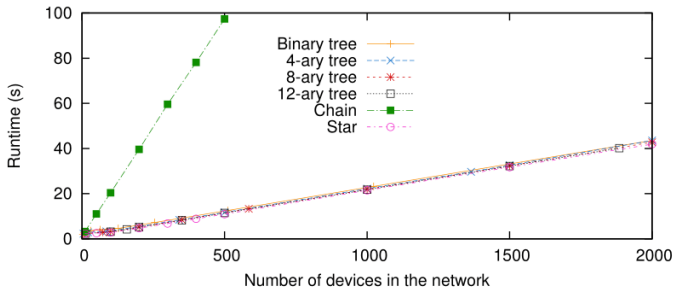
Figure 5.7: Energy consumption per device

of low-end embedded systems that are targeted by our solution.[1] For our estimations we chose the number of heartbeat intervals within t_{att} to be 20, i.e., $\#hb = 20$. Energy consumption estimations are presented in Figure 5.7. In the instantiation based on digital signatures, the energy consumption on each device D_i is quadratic in the size of the network. On the other hand, in the instantiation based on MACs, D_i's energy consumption is linear in the network size. This reduction is mainly due to the reduced communication overhead in addition to the use of computationally inexpensive MACs. Finally, Figure 5.7b shows, for the instantiation based on MACs, that the energy consumption increases with the number of neighbors per device. This increase is mainly due to the verification and re-generation of MACs on a hop-by-hop manner.

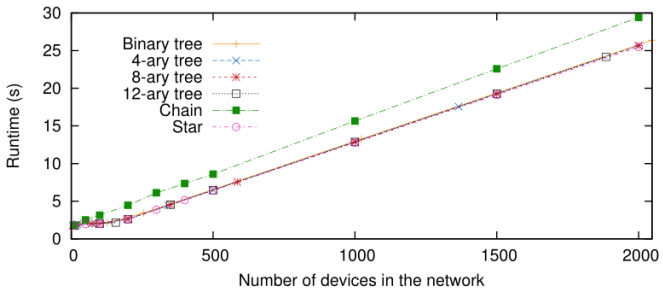
Simulation Results. We utilized the OMNeT++ [104] network simulator to assess the performance of our solution for large networks of embedded devices. Our protocols were implemented on the application layer, where we emulated cryptographic operation with delays corresponding to real measurments of their execution time on TrustLite [84]. For our simulations, the average communication rate between devices was set to 20 Kbps, which corresponds to the minimum bandwidth for ZigBee [135] – a common protocol for IoT devices. We considered three popular topologies: a star topology, a chain topology, and tree topologies with number of child nodes varying from 2 to 12. We also simulated various network sizes, which varied from 10 to 100,000. We simulated the two instantiation described in Section 5.1.2.1 and 5.1.2.3 which are based on digital signatures and MACs respectively. Figure 5.8, and 5.9 show the results of our simulations.

For both instantiations and all the aforementioned topologies the runtime of beat increases linearly with the size of the network (Figure 5.8). The instantiation based on MACs performs significantly better than that based on digital signatures. This is particularly true for chain topology. Recall that, the communication overhead of beat increases linearly with the size of the network. However, the increase in computational overhead is logarithmic for tree topologies and linear for chain and star topologies. Consequently, the performance gain from using MACs in tree topologies is significantly less than chain topology. The figure shows that the overall runtime of beat in a network of 1000 devices

1 SMART and TrustLite are only available as FPGA implementations, which tend to consume more energy than manufactured chips.

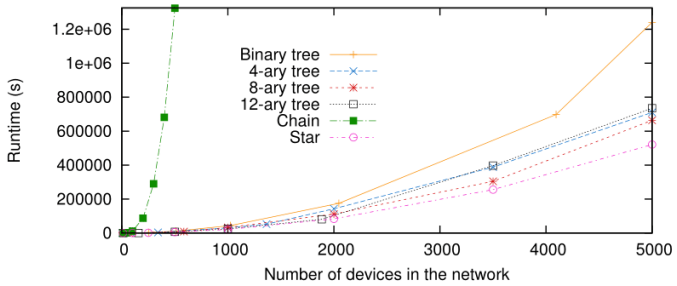


(a) Instantiation based on signatures

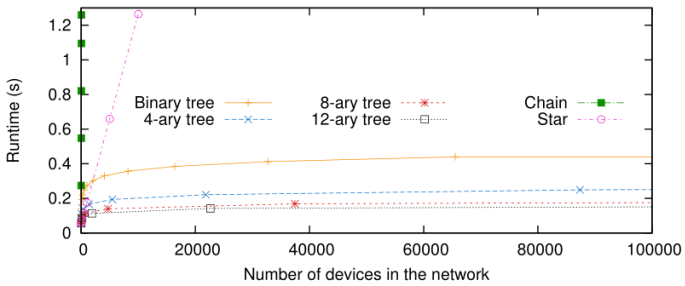


(b) Instantiation based on MACs

Figure 5.8: protocol beat



(a) Instantiation based on signatures



(b) Instantiation based on MACs

Figure 5.9: protocol collect/ attest

is less than 13 s. This runtime is significantly less than the anticipated time t_{phy} required by $\mathcal{A}$ to physically attack the hardware of a device.

The runtime of collect is quadratic in the size of the network for the instantiation based on digital signature, and the runtime of attest is logarithmic in tree toplogies and linear in star and chain toplogies for the instantiation based on MACs. This is mainly caused by the quadratic communication overhead in the signature-based instantiation. Recall that, in the instantiation based on signatures, devices are expected to send their very large logs to V, while in the instantiation based on MACs, devices only send one constant size response containing XORed MACs. Therefore, the very large difference in runtimes is mainly due to communication overhead. In the instantiation based on MACs, the runtime of attest converges to the computational overhead on involved devices, which is logarithmic in tree topologies and linear in star and chain topologies.

A quick comparison between the three topologies shows that the worst topology for collect is chain topology, where responses are always communicated through all n devices. On the other hand, protocols dominated by communication overhead (e.g., instantiation of collect based on signatures) perform best in star topology, while those dominated by computational overhead (e.g., instantiation of attest based on MACs) perform worst in such topology.

5.1.5 *Security Analysis*

The goal of our solution is to enable the verifier V to check both software and hardware integrity of all devices in a network $\mathcal{N}$, i.e., V should return $b = 1$, i.e., accept the attestation / presence report, if the adversary did not compromise the software or physically attack the hardware of any device D_i in $\mathcal{N}$. This security goal can be formalized as a security experiment $\mathbf{Exp}_{\mathcal{A}}$, where the adversary $\mathcal{A}$ can interact with every device in $\mathcal{N}$ as well as V. Recall that, $\mathcal{A}$ has full control over the communication channel between every two devices in $\mathcal{N}$, and between $\mathcal{N}$ and V, i.e., it can eavesdrop on, modify, drop, and inject arbitrary messages to any $D_i \in \mathcal{N}$ and V. $\mathcal{A}$ remotely compromises the software of one device D_s and/or exploits physical proximity to physically attack the hardware of one device D_h. At the end of the experiment, V outputs the result b of the protocol indicating whether it has accepted the report or not, following a polynomial number (in ℓ_N, ℓ_q, and ℓ_{mac}) of steps performed by $\mathcal{A}$. The output of V represents the result of this security experiment, i.e., $\mathbf{Exp}_{\mathcal{A}} = b$. In the following we provide the definition of secure collective attestation under the pre-described hybrid adversary model $\mathcal{A}_{n,n-1}$:

Definition 5.1 (Secure collective attestation under hybrid adversary)**.** *Let f be a polynomial function in ℓ_N, ℓ_q, and ℓ_{mac}. We consider a collective attestation scheme to be secure under hybrid adversary if the probability* $\Pr\left[b = 1 | \boldsymbol{Exp}_{\mathcal{A}}(1^{\ell}) = b\right]$ *is negligible in $\ell = f(\ell_N, \ell_q, \ell_{mac})$.*

Theorem 5.1 (Security of our attestation solution)**.** *The solution presented in this section is a secure collective attestation scheme (Definition 5.1) if the underlying MAC scheme is selective forgery resistant and the attestation protocol is secure under software attacks according to Definition 4.1.*

Proof sketch of Theorem 7.1. As a response to a request $Ch_1 = \{t, N, \mu_1\}$ containing a timestamp t, a nonce N, and a MAC μ_1 over t and N based on the symmetric key k_1 shared between V and D_1, the verifier receives from D_1 a message $\{resp_1, attest_1\}$. The verifier accepts and returns $b = 1$ only if $attest_1$ is a valid collective attestation report of $\mathcal{N}$ indicating that all devices in $\mathcal{N}$ are not software compromised, and $resp_1 = xor_1^n mac(k_i; 1 \| N)$. Consider the two cases where $\mathcal{A}$ (1) only compromises the software one or more devices (including D_s) without physically attacking the hardware of any device, or (2) physically attacks the hardware of up to $n-1$ devices (including D_h). It is easy to see that all possible attacks by $\mathcal{A}_{n,n-1}$ are covered by one of these cases.

We first consider the case where $\mathcal{A}$ compromises only the software of devices in $\mathcal{N}$. According to Definition 4.1, if none of the devices is physically attacked then the probability that the attestation response indicating that all devices in $\mathcal{N}$ are benign when the software of at least one device D_s is compromised is negligible in ℓ_N, ℓ_q, ℓ_{mac}.

We now consider the case where $\mathcal{A}$ also physically attacks the hardware of devices in $\mathcal{N}$. Recall that the adversary is capable of physically attacking the hardware of all but one device. Lets assume that D_i is the device that has not been physically attacked. According to beat, a heartbeat of a device D_j is accepted by D_i if it indicates by p_i the current heartbeat interval, and if it was received within the tolerance interval t_{tol}. We now calculate the lower and upper bound on the time when a received heartbeat would be accepted by D_i. Assume $timer_{hb}$ on D_i expires for the current heartbeat interval at

t_1, i.e., D_i generates and broadcasts its heartbeat hb_i at t_1. D_i also accepts all valid heartbeat for the current interval that are received within the tolerance interval $t_{tol} = t_{tr} + \delta_t$, i.e., before $t_1 + t_{tr} + \delta_t$. On the other hand, since all benign devices only generate heartbeats upon the expiry of $timer_{hb}$, correct heartbeats for current heartbeat interval are not generated earlier than $t_1 - \delta_t$. Therefore, the lower bound on the time a received heartbeat for the current interval is accepted is $t_1 - \delta_t$, and the upper bound is $t_1 + t_{tr} + \delta_t$. Similarly, the lower bound on the time a received heartbeat for the next heartbeat interval indicated by $p_i + 1$ is accepted is $t_2 - \delta_t$, and the upper bound is $t_2 + t_{tr} + \delta_t$, where $t_2 = t_1 + t_{hb}$ is when $timer_{hb}$ of the next heartbeat interval expires. Consequently, in order not to be detected as absent by D_i, a device D_j can be absent for a period of time which is no longer than:

$$t_2 + t_{tr} + \delta_t - (t_1 - \delta_t) = t_1 + t_{hb} + t_{tr} + \delta_t - t_1 + \delta_t = t_{hb} + 2 \cdot \delta_t + t_{tr}$$

Since $t_{phy} > t_{hb} + 2 \cdot \delta_t + t_{tr}$, every device D_h whose hardware is physically attacked by $\mathcal{A}$ will be detected as absent by D_i.

Indeed after physically attacking D_h, $\mathcal{A}$ will have access to all its secret keys. $\mathcal{A}$ may then try to make up for the absence of D_h by generating the missing heartbeat and broadcasting it to all device in $\mathcal{N}$ including D_i. However, it would be too late by then, as the bit OK is already set to 0 indicating the absence of at least one device ($OK = OK \wedge (\wedge_i^s present[i])$). Therefore, $\mathcal{A}$ may try to evade detection of the physical attack on D_i by one of the following ways, i.e., $\mathcal{A}$ may try to: (1) modify the code for beat or collect, (2) extract the key k_i of a device that it has not physically attacked, (3) modify the private data, e.g., OK or $present[]$ on D_i, (4) tamper with the clock of D_i to extend the time a received heartbeat is accepted. However, $\mathcal{A}$ is not capable of executing (1) since the integrity of the code for beat and collect is protected by ROM and secure boot on SMART and TrustLite respectively. Moreover, since the private data of the protocol are only accessible to the protocol code, $\mathcal{A}$ cannot execute (2) or (3). Finally, Reliable Read-Only Clock (RROC) of D_i rules out (4).

Therefore, the probability of $\mathcal{A}$ convincing V to return $b = 1$ after compromising the software of at least one device in $\mathcal{N}$ and/or physically attacking the hardware of another is negligible in ℓ_N, ℓ_q, ℓ_{mac}.

□

5.1.6 *Conclusion*

In this section we introduced a strong adversary model for collective attestation that is capable of physically attacking the hardware of a large number of devices. We then devised the first collective attestation solution that allows efficient and scalable detection of both software and physical attacks on very large networks of embedded devices. The solution we presented in this section is most suitable for centralized network. It uses absence detection for identifying devices whose hardware has been physically attacked. And, regardless of its considerable runtime overhead, it provides the grounds for developing collective attestation solutions that are secure in the presence of a pow-

erful adversary. In Section 5.2 we present another solution for detecting software and physical attacks, that is based on the results of this section. The solution presented in Section 5.2 is extremely efficient. It is geared towards autonomous decentralized networks. However, it can be extended to allow reporting to a central verifier.

5.2 UNATTENDED SCALABLE ATTESTATION OF EMBEDDED DEVICES

The solution we presented in Section 5.1 provided the ground for designing collective attestation that is capable of detecting both software and physical attacks. In particular, Section 5.1 defines the different adversary models with their varying capabilities and identifies the requirements for mitigating the most powerful adversary. It then devises a secure collective attestation solution that is capable of mitigating this powerful adversary based on absence detection. Further, in Section 5.1 we analyzed the security of the proposed solution and derived tight bounds on the time required by an adversary to physically attack the hardware of a device in order to evade detection. In order to guarantee security and provide applicability to networks where all but one device can be physically attacked, this solution required all devices to send global heartbeats to all other devices in the network. This lead to a runtime overhead that is quadratic in the size of the network, which hindered scalability to very large networks. Moreover, this solution is geared towards centralized networks that have restricted mobility, i.e., it assumes that the topology is static during the execution of the solution.

In this section, we present a second collective attestation solution for detecting software and physical attacks on large networks. While this solution imposes more assumptions regarding the capabilities of the adversary, it provides a significant performance gain, and is applicable to both autonomous and centralized networks with dynamic topology. The solution combines local absence detection and attestation of neighbors with key management in order to efficiently detect software and physical attacks while allowing device mobility. Dynamicity also requires that the outcome of protocol execution can securely flow across the network, to enable verifying the trustworthiness of mobile devices while roaming throughout the network. This is achieved through a dedicated roaming protocol. As mentioned earlier, the solution presented in this section is based on the results from Section 5.1, which defined the adversary model and identified the requirements for secure collective attestation under physical attacks.

Contribution. We investigate the security of large autonomous dynamic networks of heterogeneous embedded devices under physical attacks. We identify a realistic threat model for these networks that allows devising an efficient collective attestation solution that is capable of detecting and isolating both software and physical attacks in constant time. The solution is based on the assumptions, requirements, and time bounds established in Section 5.1. It exploits neighbors attestation, local heartbeats, key management, and a roaming protocol in order to mitigate a powerful adversary in autonomous dynamic networks. In order to demonstrate feasibility, we show how to instantiate our solution on recent security architecture for low-end embedded devices with different security features and functional capabilities: SMART [52] and TrustLite [84]. Further,

we show practicality of our solution through implementation and testing on a network testbed that constitutes six interconnected drones forming an ad hoc network. Finally, we present extensive performance evaluation based on the two instantiations, in addition to simulations of the solution in networks of up to $1,000,000$ devices demonstrating scalability. The solution allows the detection of software and physical attacks in a network formed of million devices in less than one second. This is achieved by relaying on peer attestation and presence detection, while exploiting key management and secure roaming to allow mobility.

Outline. After providing a brief overview of our solution in Section 5.2.1, we present it details in Section 5.2.2, and describe our implementation in Section 5.2.3. Performance evaluation is then presented on Section 5.2.4. Security of the solution is examined in Section 5.2.5, possible extensions are described in Section 5.2.6, and this section concludes in Section 5.2.7.

5.2.1 *Collective Attestation*

5.2.1.1 *Problem Description and System Model*

We consider a dynamic decentralized network $\mathcal{N}$ that is formed a very large number n of embedded devices. Devices are heterogeneous in term of software and hardware. Therefore, they may belong to different classes of software $C_1, \ldots, C_z$. The network operator is denoted by O. It is responsible for initializing every device D_i in $\mathcal{N}$ in a secure environment. The devices in $\mathcal{N}$ are spread over a large area both inside and outside the physical security perimeter of the network operator. Therefore, some devices can be within the adversary's grasp and are subject to physical attacks. We do not assume the existence of a central entity responsible for assessing the trustworthiness of $\mathcal{N}$. However, if such an entity exists we denote it by the verifier V. $\mathcal{N}$ may not have a routing protocol in place. However, devices in $\mathcal{N}$ should be able to communicate to their direct neighbors. Device mobility is contiguous, i.e., devices move from one neighborhood to another gradually changing their sets of neighbors, however, a device may not disappear from a neighborhood to instantaneously reappear in another. Finally, devices are assumed to be always reachable. A device may not be switched off for a long period of time. The goal of a collective attestation solution is to enable the detection and isolation of all devices whose software has been compromised or hardware has been physically attacked.

5.2.1.2 *Requirements Analysis*

Objectives. In the system model presented above there exists no central entity that attests the network and verifies the attestation results. Consequently, a collective attestation solution should provide the following properties:

- *Property #1:* Allow collective verification of the network's integrity.
- *Property #2:* Detect devices whose software has been compromised and/or hardware has been physically attacked.

- *Property #3:* Assure isolation of devices that were detected to be attacked.
- *Property #4:* Be efficient, ideally provide an overhead which is constant in the size of the network.
- *Property #5:* Allow device mobility by capturing the dynamic behavior of $\mathcal{N}$.

The satisfaction of property #1 and property #2 are the main security objective of collective attestation under physical attacks. The solution proposed in Section 5.1 aims at satisfying these properties in centralized networks with restricted mobility. Property #3 is crucial in autonomous networks where a centralized verifier is not assumed. It is achieved by combining detection of attacks with key exchange. Property #4 enables scalability to very large networks. It is achieved through peer attestation and absence detection. And, property #5 allows applicability to dynamic networks. It is enabled through a secure roaming protocol.

Adversary Model. Similar to Section 5.1, we assume that the operator O is trusted, and the verifier V, if exists, is also trusted. We assume that the adversary $\mathcal{A}$ has complete control over all communication channels, i.e., it can eavesdrop on, modify, drop, or replay any message exchanged between all devices in $\mathcal{N}$ and between any device D_i and V. Moreover, we assume that $\mathcal{A}$ is capable of executing two types of attacks: remote software attacks, whereby $\mathcal{A}$ maliciously modifies the unprotected software on any device D_i. We assume that $\mathcal{A}$ can compromise the software of all n devices in $\mathcal{N}$; and, physical attacks, whereby $\mathcal{A}$ exploits physical proximity to D_i to tamper with its hardware, modify its software and extract its secrets that are protected by hardware. We assume that $\mathcal{A}$ can physically attack the hardware of up $m < \frac{n}{2} - 1$ devices. Furthermore, we assume that $\mathcal{A}$ requires a non-negligible amount of time to physically attack the hardware of one device. We consider physical attacks that do not require turning off the device for a non-negligible amount of time, e.g., side channel attacks, to be out of scope.

Device Requirements. The design of our solution aims as satisfying all properties #1 to #5. However, in order to satisfy property #1 and property #2 every device D_i should satisfy the requirements for secure collective attestation under hybrid adversary as discussed in Section 5.1.1. In particular, every device D_i in $\mathcal{N}$ should be equipped with a Reliable Read-Only Clocks (RROC) that is not modifiable by software, and a lightweight security architecture that enables secure remote attestation, e.g., SMART [52] and TrustLite [84] (see Chapter 3). If the first requirement is not achieved, devices whose hardware is physically attacked would be able to evade detection. If the second requirement is not achieved, devices whose software is compromised and/or hardware is physically attacked would be able to evade detection.

Assumptions. Devices can have different hardware and/or software. However, all devices should satisfy the requirements for secure collective attestation under hybrid adversary specified in Section 5.1.1, i.e., devices should have synchronized RROC and a lightweight security architecture that allows secure remote attestation. We denote the maximum clock skew between any two devices by δ_t. We assume that all devices can communicate, i.e., a device can at least communicate to its neighbors. The time required

to transmit a message between two neighboring devices is upper bounded by t_{tr}. Further, we assume that all cryptographic primitives and their implementations are secure.

Observations. Let $\texttt{Nhbr}(t_a,X,Y)$ denote "devices X and Y are neighbors at time t_a", $\texttt{Met}(t_a,X,Y)$ denote "devices X and Y became neighbors at time t_a", $\texttt{Present}(t_a,X,Y)$ denote "device X believes that device Y is present in the network since deployment and until time t_a (i.e., not absent for longer than t_{phy}), $\texttt{Benign}(t_a,X,Y)$ denote "device X believes that device Y is not software compromised at t_a, $\texttt{Secure}(t_a,X,Y)$ denote "device X believes that device Y was not physically attacked since deployment and until time t_a, $\texttt{Equal}(t_a,X,Y)$ denote "device X and device Y have the same software state at time t_a, and $\texttt{Short}(t_a,t_b)$ denote "The difference between t_a and t_b is less than t_{phy}. We make the following observation:

- **Non-negligible tampering time:** Our core assumption is that an adversary requires a non-negligible amount of time to physically attack the hardware of a device, e.g., to dissemble it components and recover its cryptographic secrets. Consequently, the presence of a device D_i in the network implies its physical security.

$$\forall X\ \forall Y\ \forall t_a\ \texttt{Present}(t_a,X,Y) \rightarrow \texttt{Secure}(t_a,X,Y)$$

- **Transitivity:** Knowledge of presence, software, and physical trustworthiness is transitive, i.e., if D_i believes that D_j has always been present in the network $\mathcal{N}$ until time t_a, and D_j believes that D_k is present in $\mathcal{N}$ until t_a. Then, D_i believes that D_k is present in the $\mathcal{N}$ until t_a and is consequently not physically attacked.

$$\forall X\ \forall Y\ \forall Z\ \forall t_a\ \texttt{Present}(t_a,X,Y) \wedge \texttt{Present}(t_a,Y,Z) \rightarrow \texttt{Secure}(t_a,X,Z)$$

  Similarly, if D_i believes that D_j has always been present in the network $\mathcal{N}$ until time t_a, and D_j believes that the software of D_k is not compromised. Then, D_i believes that the software of D_k is not compromised.

$$\forall X\ \forall Y\ \forall Z\ \forall t_a\ \texttt{Present}(t_a,X,Y) \wedge \texttt{Benign}(t_a,Y,Z) \rightarrow \texttt{Benign}(t_a,X,Z)$$

- **Contiguous mobility:** Devices approach each other gradually i.e., when D_i moves toward D_j it establishes neighborhood with all devices on its path to D_j. Consequently, before D_i and D_j come within each others range, they have at least one common neighbor.

$$\begin{aligned}\forall X\ \forall Y\ \forall t_a\ \texttt{Met}(t_a,X,Y) \rightarrow \exists Z\ \exists t_b\ \exists t_c\ &\texttt{Nhbr}(t_b,X,Z)\\ &\wedge\ \texttt{Nhbr}(t_c,Z,Y) \wedge\ \texttt{Short}(t_a,t_b)\ \wedge\ \texttt{Short}(t_a,t_c)\end{aligned}$$

- **Healing:** Two similar devices D_i and D_j that have the same software configuration can be either both benign or both compromised.

$$\forall X\ \forall Y\ \forall Z\ \forall t_a\ \texttt{Equal}(t_a,Y,Z) \wedge \texttt{Benign}(t_a,X,Y) \rightarrow \texttt{Benign}(t_a,X,Z)$$

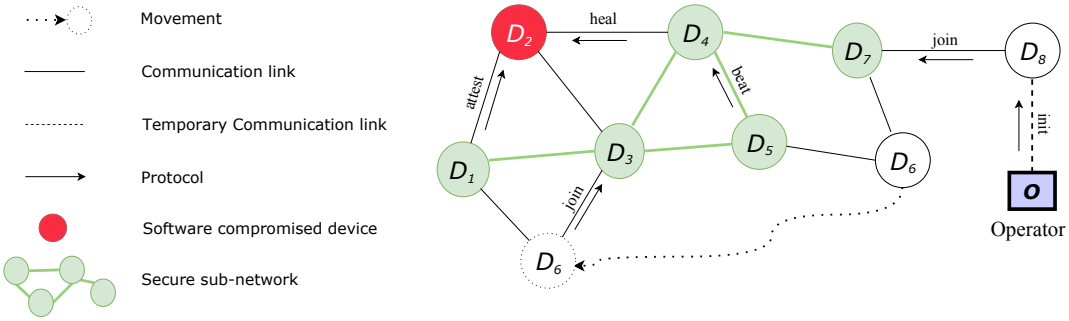


Figure 5.10: Example 8-device network: $D_1, \ldots, D_8$

Consequently, in dynamic autonomous networks, where devices only interact with their direct neighbors, securing the network against both software and physical attacks requires only local attestation/heartbeats. However, the protocols' results must be incorporated within the network's functionality. Note that, a roaming protocol would enable movement of mobile devices. Moreover, healing of device's software can be enabled by letting devices, that belong to the same software class, recover each other.

Protocol Overview. The basic idea of our solution is that devices in $\mathcal{N}$ periodically monitor the presence and the software configuration of their neighbors. If a device is detected by neighbors to be software compromised or physically attacked, this device is disconnected and isolated from the network $\mathcal{N}$. The solution also allows new devices to join, and mobile devices to move throughout $\mathcal{N}$. Finally, the compromised software of a device can be restored by a similar device using a dedicated protocol. Figure 5.10 shows an overview of our solution in a network formed of eight devices D_1 through D_8. The proposed solution consists of five protocols: initialization init, roaming join, attestation attest, heartbeat beat, and healing heal.

- init: Before deployment, each device (D_8 in Figure 5.10) is initialized by O with the cryptographic secrets necessary to execute the other protocols (e.g., a signing key pair).
- join: When a new device (D_8 in the figure) joins $\mathcal{N}$, or a mobile device changes its positions (D_6), the device executes join with all its newly established neighbors. join allows the device to (1) prove its trustworthiness to the new neighbors (e.g., through a Proof-of-non-Absence – PonA), and to share symmetric keys that allow the authentication of messages for other protocols.
- attest: At random times, each device (D_1) attests all its neighbors (e.g., D_2) through attest. Devices maintain a list of neighbors that were successfully attested. Further, each device drops the secure communication to all neighbors that failed attestation by deleting all keys shared with these neighbors.
- beat: The goal of beat is to allow devices to keep track of neighbors' presence in $\mathcal{N}$. Periodically, each device (D_5) sends a heartbeat to all its neighbors (e.g., D_4)

demonstrating its presence in $\mathcal{N}$ for the specific period of time. Devices maintain a list of neighbors that were present at the current time period. Further, each device drops the secure communication to all the neighbors that where not present, i.e., neighbors from which it did not receive a heartbeat. In response to heartbeats, devices send their neighbors Proof-of-non-Absence (PonA) tokens, which are later used by mobile devices to prove their trustworthiness to newly established neighbors.

- heal: When the software of a device (D_2) is detected to be compromised, a similar device (D_4), disinfects it through heal and restores its software to its original state.

Through attestation and absence detection, every device is capable of recording the software and hardware state of every neighboring device. Consequently, by dropping secure communication with neighbors that failed to prove their trustworthiness, our solution facilitates the formation of a securely connected sub-network of benign devices: {D_1, D_3, D_4, D_5, and D_7}, i.e., enabling isolation of malicious devices (D_2). Note that, autonomous networks might be occasionally (or periodically) visited by a trusted third party (e.g., the verifier V) in order to verify overall network trustworthiness. In Section 5.2.6 we describe an extension to our solution that allows such verification.

5.2.2 *Protocol Description*

5.2.2.1 *Notation*

Before describing the details of our solution, we introduce the used notation:

Heartbeat Interval. Time is split into intervals of uniform length denoted by heartbeat intervals. The length t_{hb} of each interval is upper bounded by the minimum time t_{phy} required by $\mathcal{A}$ to physically attack a device. Every heartbeat interval has a unique strictly increasing ID p_i, and its starting time is denoted by T_{hb}.

Device Lists. Every device D_i has two lists (1) a benign device list $\mathcal{B}_i$ where it stores the IDs of neighbors that are not software compromised, and (2) a secure device list $\mathcal{S}_i$ which stores the IDs of neighbors that are not physically attacked. Recall that $\mathcal{B}_i$ and $\mathcal{S}_i$ are maintained through continuous attestation and absence detection respectively. Additionally, D_i maintain for the current heartbeat interval p_i a present list $\mathcal{P}_i$ which stores the IDs of neighbors that are present during this interval.

Heartbeat. A heartbeat $HB_{ij} = \{\{p_i, id_i\}, \mu_{ij}\}$ is a periodical authenticated timestamp. It is generated by every D_i with ID id_i, at T_{hb} of every heartbeat interval p_i, and sent to each neighboring D_j. HB_{ij} is authenticated using a MAC μ_{ij} based on a heartbeat key k_{ij}^{beat} shared between D_i and D_j. HB_{ij} proves to D_j the presence of D_i during the current heartbeat interval p_i.

Proof-of-Secure-Enrollment (PoSE). A PoSE π_i is an authenticated token obtained by every device D_i at initialization time. π_i includes id_i, and time T_{init} when D_i was initialized by O. It is authenticated using a digital signature σ_i based on O's private key sk_O

($\pi_i = \{\{id_i, T_{\mathsf{init}}\}, \sigma_i\}$). π_i is used by D_i when first joining $\mathcal{N}$ to prove to its neighbors that it was securely enrolled.

Proof-of-non-Absence (PonA). A PonA Π_i is a set of authenticated tokens periodically collected by every D_i from all its neighbors after proving its presence to each neighbor. Each individual token $\psi_{jk/i}$ in Π_i is authenticated by D_j, using a heartbeat key k_{jk}^{beat} shared with every neighbor D_k of D_j, and is valid for the heartbeat interval indicated by p_i. Further, $\psi_{jk/i}$ indicates by $T_A^{D_i}$, the time of the last attestation of D_i by D_j (i.e., $\psi_{jk/i} = \{\{id_i, p_i, T_A^{D_i}\}, \mu_{jk/i}, id_k\}$). Π_i is used by mobile device D_i to prove its trustworthiness to newly established neighbors. In particular, every $\psi_{jk/i}$ in Π_i provides a proof that D_i was successfully attested and continuously present in $\mathcal{N}$.

5.2.2.2 *Protocol Details*

We now describe the details of the protocols involved in our solution:

Initialization. O initializes each device D_i in $\mathcal{N}$ with the following:

- A Proof-of-Secure-Enrollment – PoSE $\pi_i = \{\{id_i, T_{\mathsf{init}}\}, \sigma_i\}$, which allows D_i to prove that its software is not compromised and its hardware is not physically attacked before deployment.
- A signing key pair (sk_i, pk_i) and its corresponding public key certificate $\mathsf{cert}(pk_i)$ signed by O, in addition to the public key pk_O of O.
- A reference software configuration c_i, which determines the benign software configuration of D_i and a corresponding software configuration certificate $\mathsf{cert}(c_i)$ signed by O.

More formally, init is:

$$\mathsf{init}(c_i, 1^\ell) \rightarrow \left(\pi_i, sk_i, pk_i, \mathsf{cert}(pk_i), c_i, \mathsf{cert}(c_i), pk_O\right).$$

Roaming. Figure 5.11 shows join protocol executed when a device D_i establishes a new neighbor D_k. The protocol is formed of four modules that are executed between the two devices: (1) sharing symmetric keys, (2) verification of PonA or PoSE, (3) attestation, and (4) derivation of keys for authentication and encryption.

In particular, when a new device D_i joins, or a mobile device D_i moves within $\mathcal{N}$, D_i exploits a Key Exchange Protocol KEP based on signing key pairs (e.g., authenticated Diffie-Hellman) to share two symmetric keys with each neighboring device D_k. Namely, the two devices share (1) an attestation key k_{ik}^{attest} that is used by D_i to attest D_k and vice versa, and (2) a heartbeat key k_{ik}^{beat} that is required for authenticating heartbeats and PonA-s sent between the two devices.

D_i and D_k then:

- Exchange reference software configurations c_i and c_k and software configuration certificates: $\mathsf{cert}(c_i)$ and $\mathsf{cert}(c_k)$. This exchange allows the two devices to later attest each other when required.

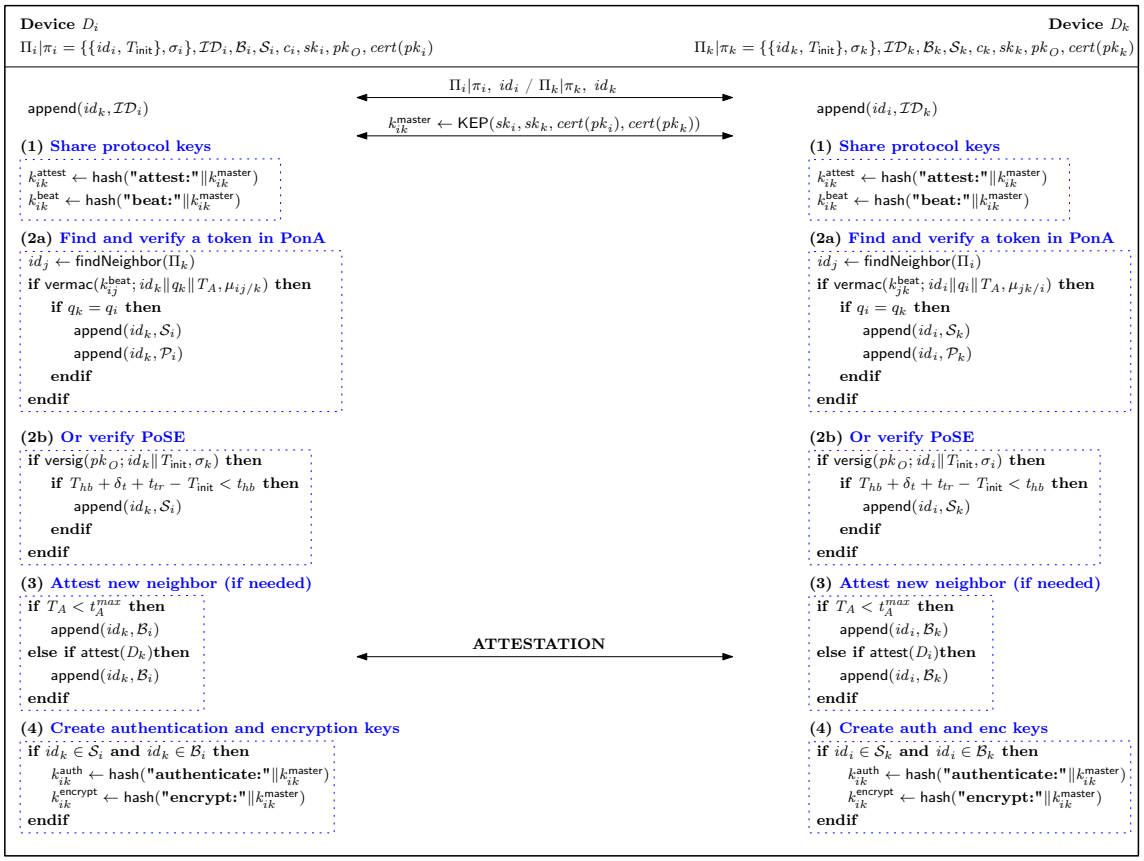


Figure 5.11: Protocol join

- Declare each other as neighbors by adding the ID of one another into their respective lists $\mathcal{ID}_i$ and $\mathcal{ID}_k$ of neighbors.
- Exchange PoSE in case of a new device, or PonA in case of a mobile devices. PoSE and PonA enable each of the devices to prove its trustworthiness to the other device.

Upon receiving a PoSE or a PonA token, each device D_k verifies this token by verifying (1) its authenticity (MAC or digital signature) and (2) freshness (checking T_{init} or p_i). Additionally, D_k uses T_{init} or $T_A^{D_i}$ to decide whether to attest D_i or not. If both token verification and attestation were successful on both D_i and D_k, the two devices then:

- Derive two new keys for authentication k_{ik}^{auth} and encryption k_{ik}^{encrypt} of messages. We denote by $\mathcal{K}_i$ the set of keys shared between D_i and all its neighbors.
- Add the IDs of each other to their respective benign $\mathcal{B}$ and secure $\mathcal{S}$ lists

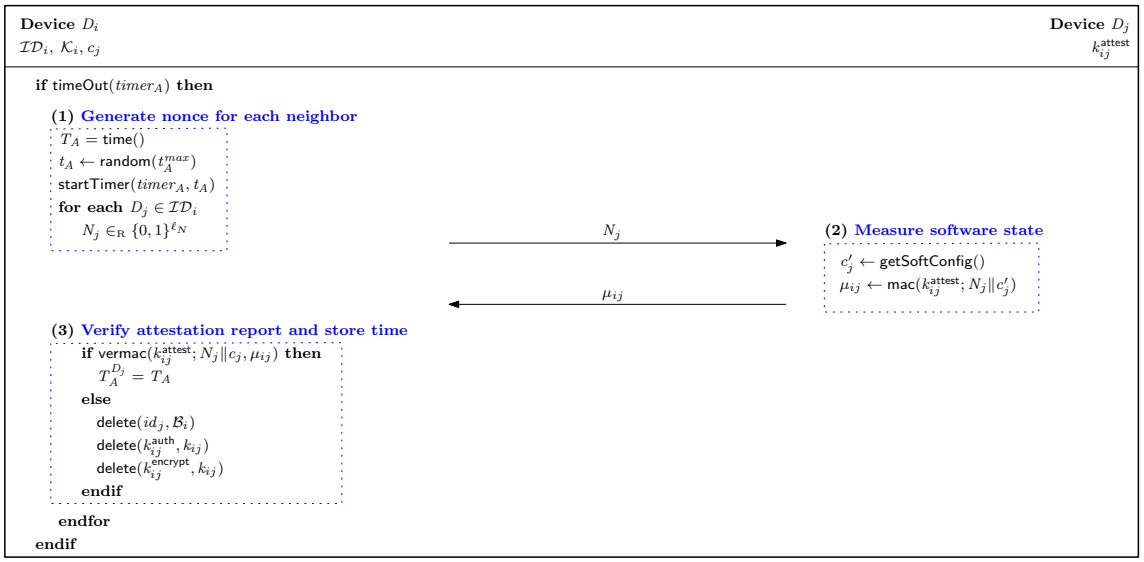


Figure 5.12: Protocol attest

As a consequence, neighboring devices that are neither software compromised nor physically attacked share authentication and encryption keys with each other through join. More formally, join is:

$$\text{join } \left[D_\text{i} : p_\text{i}, sk_\text{i}, \pi_\text{i}/\Pi_\text{i}, c_\text{i}; D_\text{k} : p_\text{k}, sk_\text{k}, \pi_\text{k}/\Pi_\text{k}, c_\text{k}; * : T_{hb}, \text{cert}(pk_\text{i}), \text{cert}(pk_\text{k}), \text{cert}(c_\text{i}), \text{cert}(c_\text{k}), pk_O\right] \rightarrow \left[D_\text{i} : k_\text{ik}, \mathcal{ID}_\text{i}, \mathcal{S}_\text{i}, \mathcal{B}_\text{i}; D_\text{k} : k_\text{ik}, \mathcal{ID}_\text{k}, \mathcal{S}_\text{k}, \mathcal{B}_\text{k}; * : c_\text{i}, c_\text{k}\right].$$

Due to contiguous mobility, join allows benign devices to connect to the network without causing false positives.

Attestation. Figure 5.12 shows attest protocol executed between two neighboring devices D_i and D_j. The protocol is formed of four modules that are executed between the two devices: (1) generation of a fresh nonce, (2) creation of software configuration, and (3) verification of an attestation report.

At random attestation times upper bounded by t_A^{max}, every device D_i attests all its neighbors. D_i sends each neighbor D_j a fresh random nonce N_j. Upon receiving N_j, D_j measures its software state and creates a MAC μ_ij over the generated software configuration and the received nonce. The MAC μ_ij is then sent to D_i as the attestation report. Using the reference software configuration c_j and the attestation key $k_\text{ij}^\text{attest}$ obtained through join, D_i verifies the attestation report μ_ij. If the verification was successful, D_i deduces that the software of D_j is benign. It then stores the attestation time to be included in any future PonA it sends to D_j. Otherwise, D_i deletes id_j from its benign list $\mathcal{B}_\text{i}$ of neighbors, and deletes the keys $k_\text{ij}^\text{encrypt}$ and k_ij^auth from the set k_ij of all keys it shares with D_j. As a consequence, devices can continuously check the integrity of their neighbors' software through attest and maintain the list $\mathcal{B}$ of benign neighbors.

As shown in Figure 5.13, a software configuration is created by D_i as the root of a Merkle Hash Tree (MHT) [99] having segments of the measured code as leaf nodes.

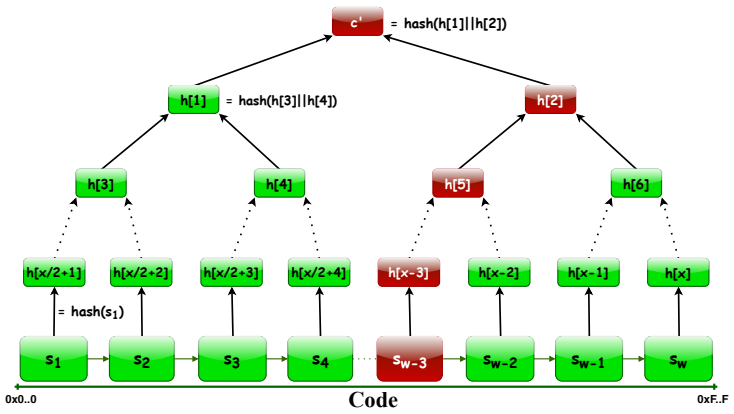


Figure 5.13: Merkle Hash Tree (MHT) of software configurations

In particular, D_i splits the measured software into w segments: $s_1, \ldots, s_w$ of equal length. It then computes hashes: $h_i[\frac{x}{2}+1], \ldots, h_i[x]$ for every segment. Next, A MHT is constructed, having $h_i[\frac{x}{2}+1], \ldots, h_i[x]$ as leaf nodes and c'_i as the root, where x denotes the number of nodes in the tree. As can be seen in the figure, a maliciously modified software segment (e.g., s_{w-3}), will lead to false hash values along the path to the root. More formally, attest is:

$$\text{attest}\left[D_i : \mathcal{ID}_i, \mathcal{K}_i, c_j; D_j : k_{ij}^{\text{attest}}; * : -\right] \rightarrow \left[D_i : T_A^{D_j}, \mathcal{B}_i; D_j : N_j\right].$$

Because of transitivity, attest ensures the software integrity of every device in the network while only requiring each device to attest its direct neighbors.

Heartbeat. Figure 5.14 shows beat protocol executed between two neighboring devices D_i and D_j. The protocol is formed of four modules that are executed between the two devices: (1) generation of a heartbeat, (2) verification of a heartbeat, (3) generation of PonA tokens, and (4) verification of MACs on PonA tokens.

Periodically, i.e., at heartbeat time T_{hb}, every device D_i generates and sends a heartbeat $HB_{ij} = \{\{p_i, id_i\}, \mu_{ij}\}$ to every neighbor D_j. HB_{ij} is authenticated using a MAC μ_{ij} based on the heartbeat key k_{ij}^{beat} shared with D_j (through join). Moreover, HB_{ij} is bound to the ID id_i of the device that generated it, and to the specific heartbeat interval p_i, during which it was generated. The goal of HB_{ij} is to prove to D_j the presence of D_i in $\mathcal{N}$ during the heartbeat interval p_i. When D_i receives HB_{ij}, it verifies (1) its authenticity by verifying μ_{ij}; and (2) its freshness according to checkTime algorithm shown in Figure 5.15. checkTime allows verifying that HB_{ij} is generated for the current interval p_i, and is received in the correct time, i.e., within the tolerance interval t_{tol} around T_{hb}. The goal of t_{tol} is allow tolerating transmission delays and clock drifts between devices.

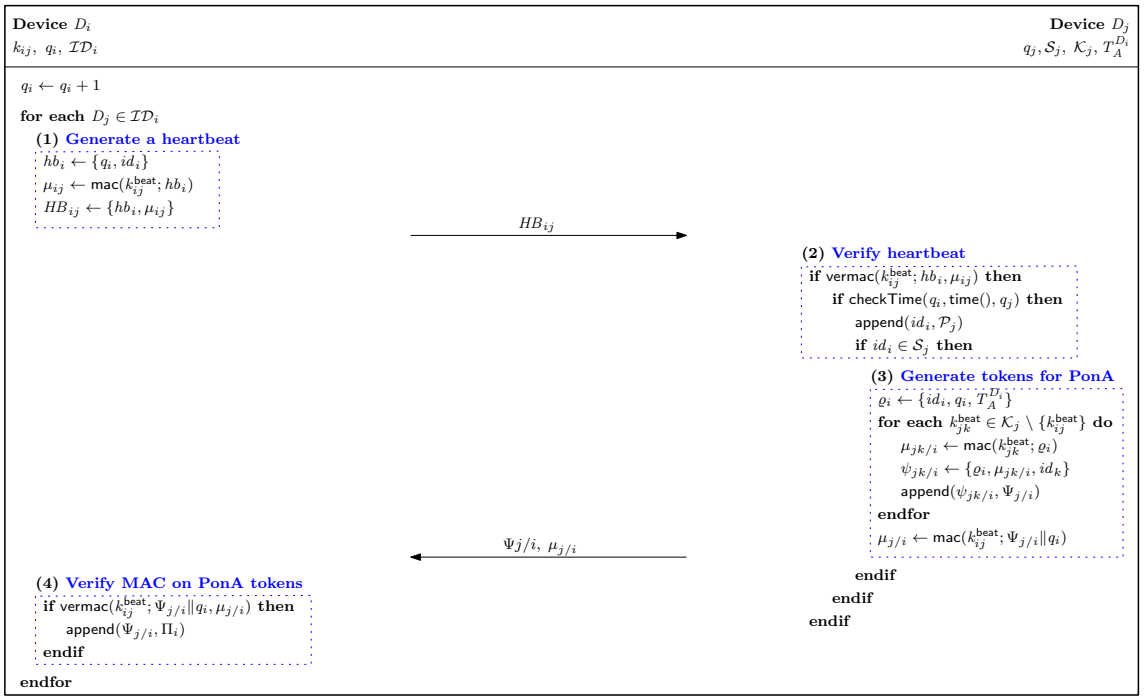


Figure 5.14: Protocol beat

Figure 5.15 : checkTime **on** D_j

if $p_i = p_j \wedge T_{hb} < t < T_{hb} + \delta_t + t_{tr}$ **then**
  return true
else if $p_i = p_j + 1 \wedge T_{hb} - \delta_t < t < T_{hb}$ **then**
  return true
else
  return false
end

If verification is successful, D_j adds the ID id_i of D_j to its list $\mathcal{P}_j$ of present devices. D_j then creates a tuple $\rho_i = \{id_i, p_i, T_A^{D_i}\}$, authenticates it with MACs based on all the symmetric keys k_{jk}^{beat} it shares with every neighboring device D_k, i.e., generating the tokens $\psi_{jk/i}$. D_j sends the set of tokens to D_i, which bundles them with all other tokens received from neighbors forming a PonA Π_i.

When t_{tol} ends, every device D_j compares the IDs in its list $\mathcal{S}_j$ of secure neighbors to those in the list $\mathcal{P}_j$ of present neighbors. D_j deletes from $\mathcal{S}_j$ every ID id_i that is not in $\mathcal{P}_j$. It also deletes the keys $k_{ij}^{\mathsf{encrypt}}$ and k_{ij}^{auth} from the set k_{ij} of all keys shared with D_i.

As a consequence, devices can continuously check the integrity of their neighbors' hardware through beat and maintain the list $\mathcal{S}$ of secure neighbors. Additionally, every secure device D_i acquires a PonA Π_i which enables it to roam throughout the network and establish new neighbors through join.

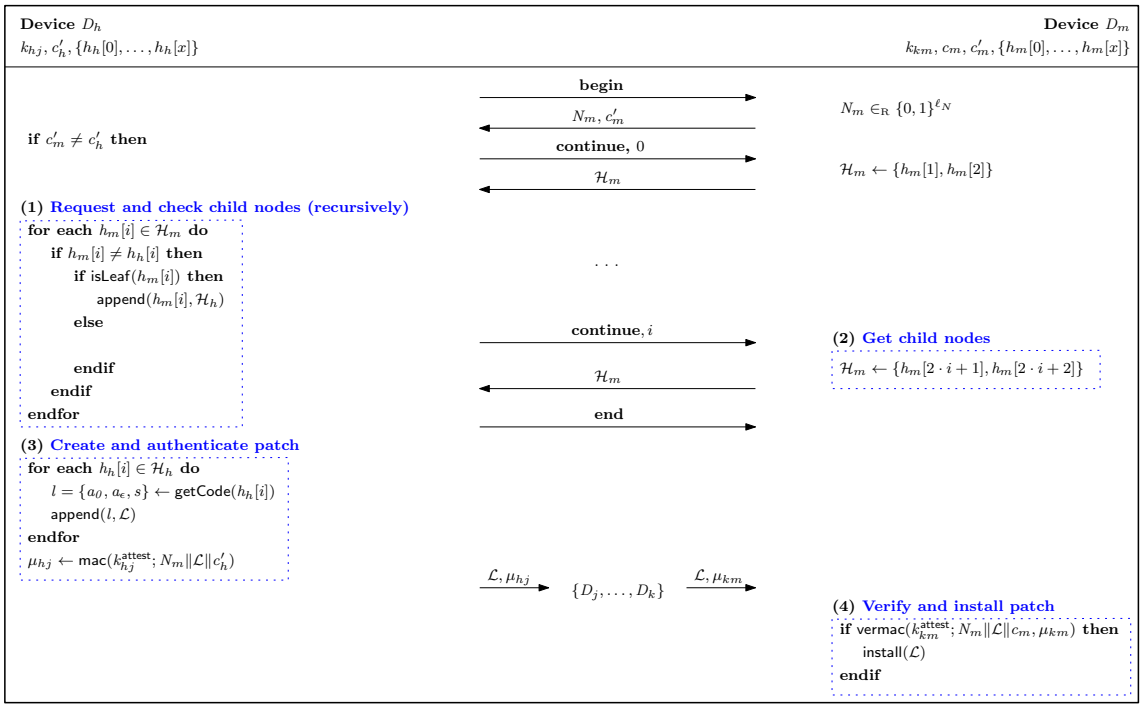


Figure 5.16: Protocol heal

Let $\mathcal{K}_j$ denote the set of symmetric keys shared between D_j and all its neighboring devices, and let $\Psi_{j/i}$ denote the set of all individual tokes $\psi_{jk/i}$ created by D_j using every symmetric key k_{jk} in$\mathcal{K}_j$. The protocol is formally:

$$\mathsf{beat}\left[D_i : \mathcal{ID}_i, p_i, k_{ij}; D_j : p_j, \mathcal{S}_j, \mathcal{K}_j, T_A^{D_i}; * : T_{hb}\right] \rightarrow \left[D_i : \Psi_{j/i}; D_j : \mathcal{P}_j\right].$$

Because of transitivity and non-negligible tampering time, beat ensures the hardware integrity of every device in the network while only requiring each device to monitor the presence of its direct neighbors.

Healing. When a device D_i detects through attest that the software of a neighbor D_m is compromised, it starts searching for a healer device D_h which is similar to D_m, i.e., has the same software configuration. D_i sends D_m's reference software configuration c_m to all neighbors along with a fresh nonce N and a constant Time-to-Live (TTL). When the tuple is received by a neighbor D_j, It verifies that (1) TTL is not equal to zero and (2) received c_m is not equal to its own reference software configuration c_j. If the checks are successful, D_j decrements TTL and forwards the tuple to its neighbors. As a consequence, c_m is flooded across the network until it is received by a healer device D_h or TTL is exceeded.

On the other hand, when D_h receives a reference software configuration c_m which matches its own reference software configuration c_h, it replies to the sending neighbor D_v with its own ID id_h, its software configuration c'_h, and a MAC over both and the received nonce based on the attestation key shared with that D_v. D_h's reply propagates

in reverse path until it is received and verified by D_i. Authenticity of the reply is based on pairwise shared symmetric keys.

heal is then initiated by D_h through sending D_m a protocol message **begin**. This protocol is formed of four modules that are executed between D_h and D_m: (1) validation MHT child nodes, (2) generation of child nodes, (3) creation of a patch, and (4) installation of the patch. The details of heal are shown in Figure 5.16. Note that, the protocol messages exchanged between the two devices may be routed through D_v using the new route established by broadcasting D_m's configuration. These messages may also be routed based on an existing routing protocol. Additionally, in order to mitigate Denial of Service (DoS) attacks on heal, D_h and D_c may authenticate messages in a hop-by-hop manner.

When D_m receives **begin**, it replies to D_h with its software configuration c'_m and a fresh nonce N_m. D_h then checks if c'_c is equal to its own software configuration c'_h. If the check fails, D_h replies with a protocol message **continue** requesting from D_m the child nodes $h_c[0]$ and $h_c[1]$ of c'_c in the Merkle Hash Tree (MHT). Next, D_h recursively requests the child nodes of every node that is not equal to its reference value in D_h's tree. When leaf nodes are reached, D_h bundles code segments corresponding to modified nodes into a single *patch* $\mathcal{L}$. The patch is then authenticated (based on hop-by-hop) MACs, and sent back to D_m. D_m verifies $\mathcal{L}$ and installs it, i.e., replaces malicious code segments indicated by $\mathcal{L}$ with the benign code from $\mathcal{L}$. Formally:

$$\begin{aligned}\mathsf{heal}\ \big[&D_h : k_{hj}, c'_h, \{h_h[0], \ldots, h_h[x]\};\\ &D_m : k_{km}, c_m, c'_m, \{h_m[0], \ldots, h_m[x]\}; * : -\big] \rightarrow \big[D_h : N_m; D_m : \mathcal{L}\big].\end{aligned}$$

Healing allows devices to recover from software compromise by installing patches from benign devices that have the same software configuration.

5.2.3 *Implementation*

We present two instantiations of this solution on top of SMART [52] and TrustLite [84] security architectures (see Chapter 3). Recall that, the two architectures we chose provide strong security guarantees for remote attestation based on minimal features in hardware, i.e., a small amount of Read-Only Memory (ROM) and a simple Memory Protection Unit (MPU).

Implementation on SMART. The instantiation of our solution on SMART [52] requires the same modifications to SMART's architecture presented in Section 5.2.3, i.e., extending MPU to control access to a small amount of rewritable memory. We require secure rewritable memory to store private data of both attestation, e.g., attestation time, and absence detection, e.g., present list. For this instantiation we stored in ROM of each device D_i the program code, i.e., the code responsible for the execution of join, attest, beat, and heal. We also store in ROM of each device D_i the signing key sk_i. The integrity of the code and the signing key is then ensured through the emutability of ROM. Moreover, we store the short term ST_i and long term LT_i data of the protocols, e.g., symmetric

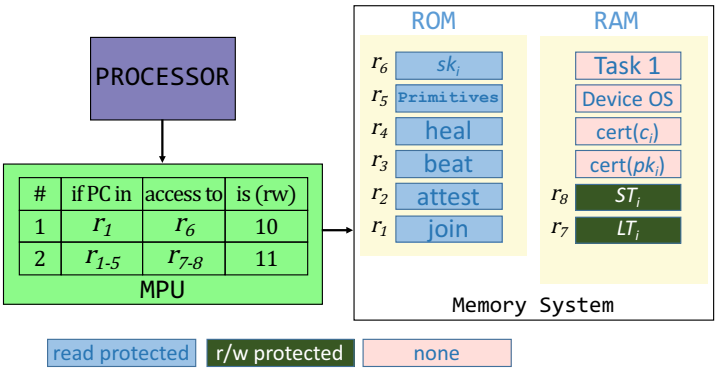


Figure 5.17: Implementation based on SMART [52]

keys $k_{ij}, \cdots, k_{ik}$ shared with neighboring devices, in the rewritable memory of every device D_i since this list could be updated during the lifetime of D_i. Our implementation on SMART is shown in Figure 5.17, where we denote rewritable memory by RAM. SMART's MPU ensures that secret data are only accessible to unmodified program code in ROM that requires access to this data. For example, rule #1 ensures that only join has read access to the secret key sk_i and rule #2 ensures that only join, attest, beat, and heal have read and write access to short term ST_i and long term LT_i protocol data.

Implementation on TrustLite. Our solution is implemented as trustlets on the TrustLite security architecture [84] (see Chapter 3). More precisely, we implemented each of the protocols join, attest, beat, and heal as a single independent trustlet on device D_i. Our implementation is shown in Figure 5.18. Trustlite ensures the software integrity of each of these protocols via the secure boot component SecureBoot on D_i. Further, as in SMART, the MPU of TrustLite ensures that secret data of D_i is only accessible to appropriate trustlets. For example rule #1 ensures that SecureBoot has exclusive read access to the memory housing the program code of join, attest, beat, and heal, rule #2 ensures that only join have read access to sk_i, and rule #3 ensure that only join, attest, beat, and heal have read and write access to short term ST_i and long term LT_i protocol data.

Figure 5.19 : authenticate **on** D_i

if $id_j \in \mathcal{S}_i$ ***and*** $id_j \in \mathcal{B}_i$ **then**
  return $\mathsf{mac}(k_{ij}^{\mathsf{auth}}; m)$
end

Figure 5.20 : verify **on** D_i

if $id_j \in \mathcal{S}_i$ ***and*** $id_j \in \mathcal{B}_i$ **then**
  return $\mathsf{vermac}(k_{ij}^{\mathsf{auth}}; m, \mu)$
end

Functionality. In order to bind attestation and absence detection to functionality and allow isolation of malicious devices, we added to our instantiation a new module denoted by `Primitives`. `Primitives` provide cryptographic primitives as services to all other software on the device, i.e., authentication and encryption, without granting access to secret keys to untrusted software. An examples for authentication is shown by authenticate and verify in Figure 5.19 and 5.20 respectively. Before executing authentication or encryption on D_i, the module ensures that the target device D_j is neither software compromised nor physically attacked by checking its ID id_j in $\mathcal{B}_i$ and $\mathcal{S}_i$ respectively.

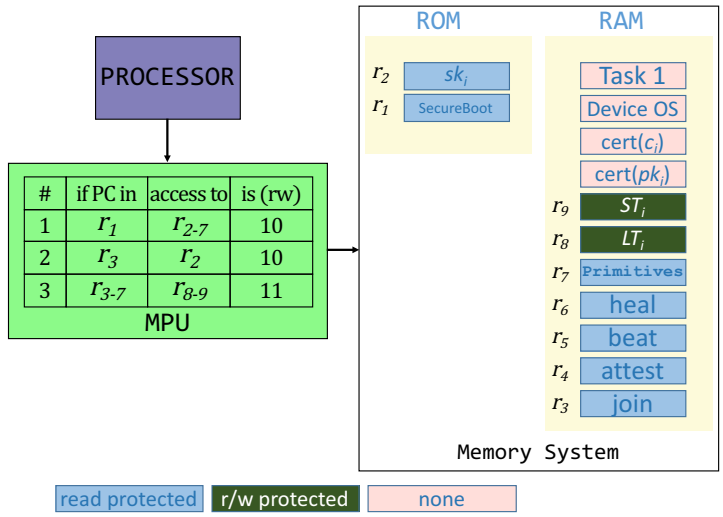


Figure 5.18: Implementation based on TrustLite [84]

5.2.4 *Performance Evaluation*

We assess performance of our solution in terms of computational, communication, memory, and energy costs. We further present simulations results for networks of up to 1,000,000 devices. This performance evaluation is based on our implementation in Section 5.2.3.

Computation Cost. Cryptographic operations, such as MAC or digital signature generation, constitute the major part of the computation cost. Let g_i denote the number of neighbors of every device D_i. Every device D_i creates $5g_i$ MACs, verifies g_i MACs, executes g_i Key Exchange Protocols KEP, and verifies g_i additional MACs (for mobile devices) or g_i digital signatures (for new devices) during join. D_i creates g_i MACs (as prover) and verifies g_i MACs (as verifier) during attest, and creates $(g_i + 2) \cdot g_i$ MACs and verifies $2g_i$ MACs during beat. Finally during heal the healer device D_h creates 1 MAC and the malicious device D_m verifies 1 MAC.

Communication Cost. We used HMAC based on SHA-1 as our MAC implementation, and ECDSA as our digital signature scheme, i.e., $\ell_{mac} = 160$ and $\ell_{sign} = 320$. We further used a 64 bit timestamp and chose $\ell_N = 160$ and $\ell_p = 64$. As a consequence, nonces and MACs are 20 Bytes each. The variables T_{init}, p, and id are 8 Bytes each. And, digital signatures are 40 Bytes each. During join, each device D_i has a communication overhead, which is upper bounded by sending and receiving $(44g_i^2 + 48) \cdot g_i$ Bytes (for mobile devices), or sending and receiving $112g_i$ Bytes (for new devices). During attest, D_i has a communication overhead, which is upper bounded by receiving $20g_i$ Bytes and sending $20g_i$ Bytes. And during beat, D_i has a communication overhead, which is upper bounded by receiving $(44g_i + 20) \cdot g_i$ Bytes and sending $36g_i$ Bytes. The communication overhead of heal depends on different variables including the size of the software, the

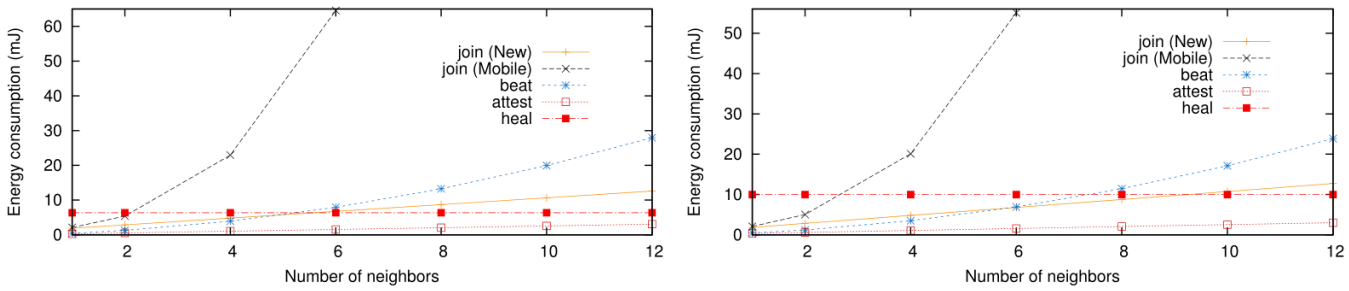
number and size of software segments, and the number of malicious segments. This overhead will not be discussed here.

Memory Cost. Every device D_i in $\mathcal{N}$ should store the following: (1) an authentication key pair (sk_i, pk_i) and the corresponding identity certificate $\text{cert}(pk_i)$; (2) a software configuration c_i and the corresponding software configuration certificate $\text{cert}(c_i)$; (3) a set of attestation and heartbeat keys that it shares with neighboring devices – $\mathcal{K}_i$; and (4) the sets $\mathcal{B}_i$, $\mathcal{S}_i$, and $\mathcal{P}_i$ of IDs of benign, secure, and present neighbors respectively. The memory cost of D_i is $36 \cdot g^2 + 56 \cdot g + 228$ Bytes, where an amount of $56 \cdot g + 80$ Bytes is read- and/or write-protected.

Table 5.1: Runtime of `Primitives`

Run-time at 48 MHz TrustLite [84] (µs) for 64 Bytes messages			
authenticate	verify	encrypt	decrypt
320	320	460	870

Runtime. We present the runtime of `Primitives` which allows authentication and encryption of messages exchanged between devices. As shown in Table 5.1 authenticating and encrypting 64 Bytes of data requires 780 µs on TrustLite. The time required to verify and decrypt 64 Bytes is 1,190 µs.



(a) MICAz (b) TelosB

Figure 5.21: Energy consumption per device

Energy Cost. We estimated the energy consumption of our solution based on the energy costs of communication and cryptographic operations reported for two sensor nodes: MICAz and TelosB [47], which belong to the same class of low-end embedded systems that we target.[2] For our estimation we excluded the energy required for generating a

2 SMART and TrustLite are only available as FPGA implementations, which tend to consume more energy than manufactured chips.

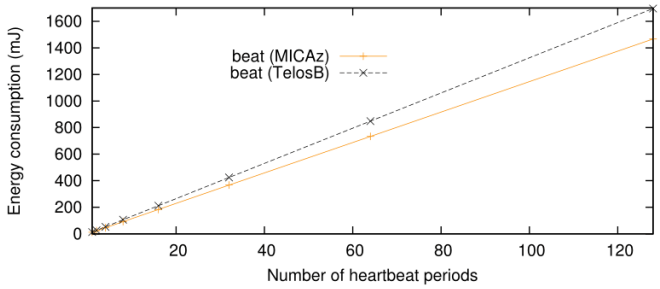
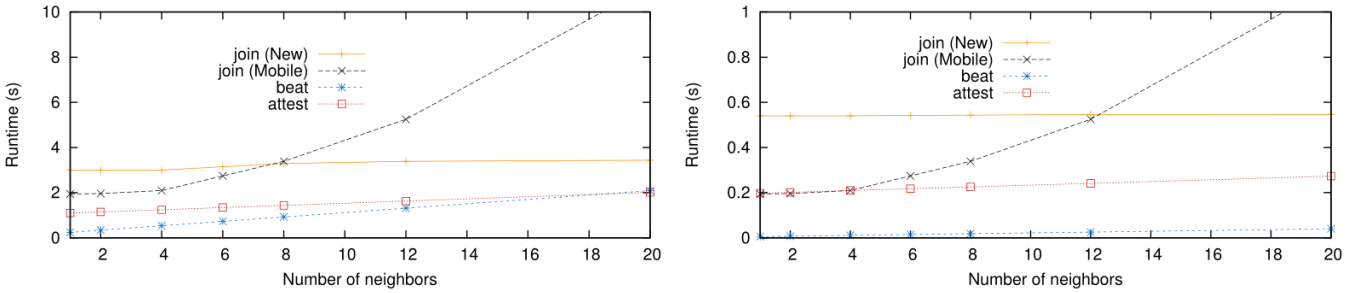


Figure 5.22: Energy consumption of beat

software configuration and for executing KEP, which would vary depending on the used protocol. Energy consumption estimations are presented in Figure 5.21, and 5.22.

The energy consumption on each device D_i increases linearly with the number of neighbors g_i of this device for all protocols except join for a mobile device. For this protocol the energy consumption is cubic in the number of neighbors. The main reason for this is the quadratic size of Proofs-of-non-Absence (PonA) in terms of number of neighbors. Note that, the energy consumed in all of the protocols is constant in network size. It can be for a MICAz device with 4 neighbors as low as 1, 5, and 20 mJ for attest, join (for a new device) or beat, and join (for a mobile device) receptively.

Finally, Figure 5.22 shows the energy consumption for both MICAz and TelosB sensor nodes with 8 neighbors as function of the number of heartbeat intervals that elapsed within a specific time period. The figure shows that the energy consumption of beat on D_i is linear in the number heartbeat intervals. Note the, heartbeat interval depends on the minimum amount of time required by the adversary to physically attack a device. Thus, it presents a trade-off between performance and security. A more accurate detection of physical attacks is provided by shorter heartbeat intervals which requires more energy.

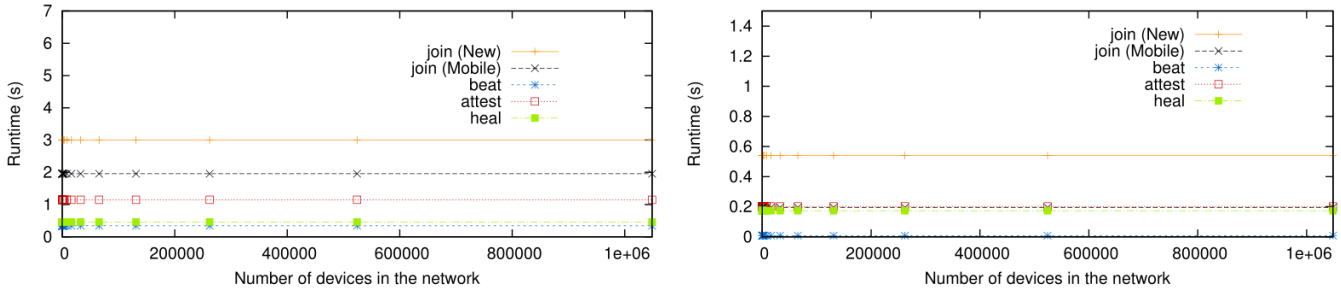


(a) SMART [52]

(b) TrustLite [84]

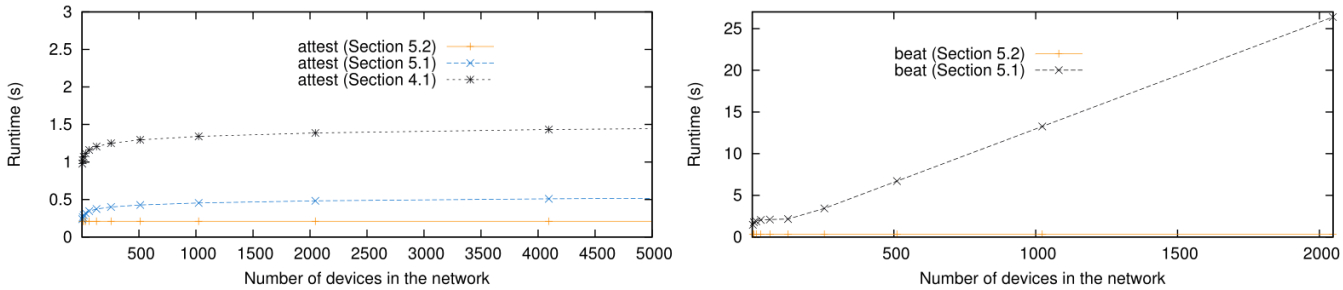
Figure 5.23: Performance of all protocols

Simulation Results. We used the OMNeT++ [104] network simulator to assess the performance of our solutions for large networks of embedded devices. Our protocols were implemented on the application layer, where cryptographic operation were emulated



(a) SMART [52] (b) TrustLite [84]

Figure 5.24: Performance in terms of network size



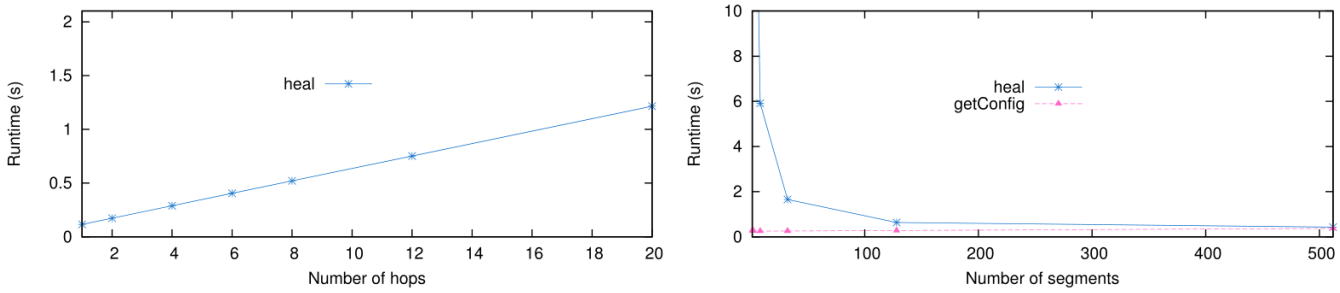
(a) vs. SEDA (b) vs. DARPA

Figure 5.25: Comparison to other solutions based on TrustLite [84]

with delays corresponding to real measurments of their execution time on SMART [52] and TrustLite [84]. For our simulations, the average communication rate between devices was set to 250 Kbps, which corresponds to the defined bandwidth for ZigBee [135]. We simulated various network sizes, which varied from 10 to 1,000,000. We also varied the number of neighbors from 2 to 20. We assume that the size of attested memory on each device is 100 KB. The execution time for KEP was excluded from our evaluation since KEP is required regardless of our solution and its overhead depends on the specific protocol used. Figure 5.23, 5.24, 5.25, and 5.26 show the results of our simulations.

As shown in Figure 5.23 the runtime of beat, attest and join (for a new device) increases linearly with the number of neighbors per device, while the runtime of join (for a mobile device) is quadratic in this number. This quadratic overhead is mainly caused by the quadratic size of PonA in terms of number of neighbors. However, as shown in Figure 5.24 the runtime of all protocols is constant in the network size.

Further, the comparison to the solutions presented in Section 4.1 and 5.1 (Figure 5.23a and 5.23b respectively), shows that the solution presented in this section performs better than both. In particular, the attest protocol presented in this section has constant overhead in the size of the network in comparison to the logarithmic overhead of attest in Section 4.1 and Section 5.1. Similarly, the beat protocol in this section has a constant overhead in comparison to the linear overhead of beat in Section 5.1.



(a) Per hops (b) Per segments

Figure 5.26: Performance of heal on TrustLite [84]

Finally, Figure 5.26 shows that the runtime of heal is linear in the number of hops between the healer D_h and the malicious device D_m, and logarithmic in the number software segments (Figure 5.26a and 5.26b respectively). Note that, Figure 5.26a assumes the code is split into 128 equal segments, and Figure 5.26b assumes that D_h and D_m are separated by 10 hops. Figure 5.26b also shows that the time required to generate a software configuration (i.e., getConfig) based on Merkle Hash Tree (MHT) increases linearly with the number of software segments.

5.2.5 *Security Analysis*

Recall that goal of a collective attestation solution is to isolate malicious (i.e., devices whose software is compromised and/or hardware is physically attacked) from benign devices in the network $\mathcal{N}$. This goal can be formalized as a security experiment $\mathbf{Exp}_{\mathcal{A}}$, where the adversary $\mathcal{A}$ can interact with every device in $\mathcal{N}$. $\mathcal{A}$ has full control over the communication channel between every two devices in $\mathcal{N}$, i.e., it can eavesdrop on, modify, drop, and inject arbitrary messages to any $D_i \in \mathcal{N}$. $\mathcal{A}$ maliciously modifies the software configuration and/or physically attacks the hardware of a device D_m. At the end of the experiment, one benign D_b outputs its decision b indicating whether it accepts to securely communicate to the malicious device D_m, following a polynomial number (in ℓ_{mac}, ℓ_{sign}, ℓ_N, ℓ_p, and ℓ_t) of steps performed by $\mathcal{A}$. The output of D_b represents the result of this security experiment, i.e., $\mathbf{Exp}_{\mathcal{A}} = b$. In the following we provide the definition of secure collective attestation under the pre-described model:

Definition 5.2 (Secure collective attestation in autonomous systems)**.** *Let f be a polynomial function in ℓ_{mac}, ℓ_{sign}, ℓ_N, ℓ_p, and ℓ_t. We consider a collective attestation scheme for autonomous systems to be secure under physical attacks if the probability* $\Pr\left[b = 1 | \boldsymbol{Exp}_{\mathcal{A}}(1^{\ell}) = b\right]$ *is negligible in* $\ell = f(\ell_{mac}, \ell_{sign}, \ell_N, \ell_p, \ell_t)$.

Theorem 5.2 (Security of our solution)**.** *The attestation solution presented in this section is a secure collective attestation scheme (Definition 5.2) if the underlying MAC and signature schemes are selective forgery resistant.*

Proof sketch of Theorem 5.2. D_b returns $b = 1$, only if (1) one of its existing neighbors D_e always sends correct heartbeats at correct times, and is successfully attested; or (2) a new or a mobile device D_n proves its trustworthiness through correct Proof of Secure Enrollment (PoSE) or Proof of non Absence (PonA) respectively. Consider the following four strategies through which $\mathcal{A}$ tries to add a malicious device D_m to $\mathcal{N}$ by attacking: beat, attest, join for a mobile device, or join for a new device. It is easy to see that all possible attacks by $\mathcal{A}$ trying to add D_m to $\mathcal{N}$ is covered by a combination of these attack strategies.

We first consider the first strategy where $\mathcal{A}$ attacks beat. Physically attacking the hardware of D_m requires a non-negligible amount of time that is greater than the heartbeat tolerance interval (see Section 5.1.5). Therefore, all neighbors of D_m will be missing at least one heartbeat of D_m and will hereby record it as physically attacked. In order to avoid this, $\mathcal{A}$ may either try to forge or replay a heartbeat while D_m's hardware is being physically attacked. However, these attacks are negligible in ℓ_{mac}, ℓ_p, and ℓ_t.

We now consider the case where $\mathcal{A}$ attacks attest. Since t_A^{max} is a strict upper bound on the time interval between two executions of attestation of neighbors, A software compromised device D_m will be detected and isolated from $\mathcal{N}$ within t_A^{max}. In order to avoid this, $\mathcal{A}$ may try to forge or replay a valid attestation response of D_m. However, these attacks are negligible in ℓ_{mac} and ℓ_N.

Next we consider the case where $\mathcal{A}$ attacks join (for a mobile device). Two devices D_m and D_b can become neighbors through join (for a mobile device) after exchanging PonAs Π_m and Π_b, only if Π_m (resp. Π_b) contains a token that proves the presence of D_m (resp. D_b) in $\mathcal{N}$ for all previous heartbeat intervals. The tokens should also indicate a recent successful attestation of each device. Consequently, a mobile device D_m, whose software has been compromised or hardware has been physically attacked, would not be able to reconnect to $\mathcal{N}$. In order to overcome this, $\mathcal{A}$ may try to forge or replay such a token generated by one of D_m's neighbors. However, these attacks are negligible in ℓ_{mac}, ℓ_p, and ℓ_t.

Finally, we consider the case where $\mathcal{A}$ attacks join (for a new device). Two devices D_m and D_b can become neighbors through join (for a new device) after exchanging PoSE π_m and PonA Π_b, only if π_m (resp. Π_b) proves trustworthiness of D_m's (resp. D_b's) software and hardware. Consequently, a new device D_m, whose software has been compromised or hardware has been physically attacked, would not be able to join $\mathcal{N}$. Similarly, even if D_m is not malicious, it cannot be added as a bridge between malicious and benign devices in $\mathcal{N}$. In order to overcome this, $\mathcal{A}$ may try to (1) forge or replay a PoSE, or (2) forge or replay a token in PonA. However, these attacks are negligible in (1) ℓ_{sign} and ℓ_t; and (2) ℓ_{mac}, ℓ_p, and ℓ_t respectively.

Therefore, the probability of $\mathcal{A}$ convincing a benign device D_b to accept to securely communicate to a malicious device D_m and return $b = 1$ is negligible in ℓ_{mac}, ℓ_{sign}, ℓ_N, ℓ_p, and ℓ_t. □

5.2.6 *Protocol Extensions*

In this section we present parameter tuning and extensions to our solution that: (1) reduce false positive rates, caused by incorrectly counting an absent node as malicious. This can occur because of battery depletion, system failure followed by reboot, or temporary unreachability; (2) remove some previous assumptions, that might not be realistic (e.g., contiguous mobility, connected network); (3) offers new features required in some scenarios (i.e., reporting to an external verifier, hereby making the solution presented in this section a superset of collective attestation solutions presented earlier); and (4) supports partitioning and allow healing (i.e., re-connection) of partitions by network operator O.

Proof-of-non-Absence (PonA) Verification. In some network settings, it is reasonable to assume contiguous mobility (as described in Section 5.2.1). However, in scenarios where devices are sparsely distributed over a large area, this assumption no longer holds. Since during join every D_i expects to find in Proof-of-non-Absence (PonA) Π_j received from a new direct neighbor D_j, an individual token $\psi_{ik/j}$, which it can verify. Applying our solution in such sparse networks will lead to many benign devices being mistakenly counted as malicious – false positives.

For this reason, we propose a PonA verification protocol, that allows devices to check the validity of a PonA even it does not include a token authenticated by a direct neighbor. When D_i fails to find a direct neighbor in Π_j, it sends to all its neighbors a request containing a constant (protocol specific) Time-to-Live (TTL), and Π_j. The request is flooded until TTL is exceeded or D_k is found, whose id id_k is in Π_j. D_k verifies the corresponding individual token $\psi_{kl/j}$ as in join. If verification succeeds, D_k creates a new token for D_j and sends it back to D_i. The token is authenticated hop-by-hop, using MACs based on symmetric keys shared between neighbors. Consequently, the new token is received and can be verified by D_i, thus, enabling secure communication between D_i and D_j.

Collection. When $\mathcal{N}$ is attended by a trusted party (e.g., the verifier V). V runs collect. The main goal of collect is to gather the number (or ids) of devices belonging to different device classes $C_1, \ldots, C_z$ in $\mathcal{N}$. V initiates collect by choosing an arbitrary device D_1 and sending it a random nonce N, a session identifier q, and the set of classes $C_u, \ldots, C_v$ it is interested in. D_1 then forwards this to all its neighbors, which in turn forward it to their neighbors and so forth, until it is received by every device in $\mathcal{N}$. Note that, q is necessary to avoid overcounting, and is used for the construction of a spanning tree rooted at D_1.

Starting at leaf nodes, each D_i sends to its parent node D_j a response $resp_i$, which contains the number r_t (or ids) of the devices in its subtree that belong to each class C_t in $\{C_u, \ldots, C_v\}$. The response is authenticated using a MAC (μ_{ij}) based on symmetric key k_{ij}^{attest}. D_j, in turn, verifies authenticity of the received responses, accumulates them, and sends them to its parent. The final response $resp_1$ is then generated by D_1, authenticated using signature σ_1 (based on its secret key sk_1), and forwarded to V. After verifying σ_1, V learns the number r_t (or ids) of benign devices in each class C_t, i.e., devices whose

software is not compromised and hardware is not physically attacked. The result is only accepted by V, if $\sum_{t=u}^{v} r_t > m$. Formally:

$$\mathsf{collect}\left[V : C_t, \ldots, C_u N, q; D_1 : \mathcal{K}_1, sk_1, ; * : \mathrm{cert}(pk_1)\right] \rightarrow \left[V : resp_1; D_1 : N, q, C_t, \ldots, C_u\right].$$

Similar to collective attestation solutions presented earlier, collect reports the status of the network to a trusted third party. However, collect has three differences to these solutions:

- Efficient detection of physical attacks: In addition to detecting remote software attacks, collect also enables V to detect physical attacks. However, unlike Section 5.1 the heartbeat protocol it is based on imposes constant overhead in the size of the network.
- Lower overhead and mobility: During the execution of collect, devices are not required to measure their software configuration, which considerably reduces runtime as well as energy consumption of the protocol. Reducing reporting time is particularly important since the topology is assumed static during the execution of collect.
- Resiliency to Denial of Service (DoS) attacks: Since devices do not perform full-blown attestation during collect, computational DoS on the network is no longer possible.

Finally, in case of network partitioning, V may run collect with different network partitions in order to detect the overall number of benign device in the whole network. Similar to what is described above, V considers every partition, where the number of benign devices is greater than m, as a benign partition.

Partition Healing. Having attested the network via collect, V becomes aware of partitions in $\mathcal{N}$. Thus, in order to heal the network's partitions (i.e., reconnect them), V runs a partition healing protocol. V simply broadcasts for every D_i in each benign partition (e.g., through the same arbitrary device D_1 used for collect) a Proof-of-Secure-Enrollment (PoSE) ($\pi_i = \{\{id_i, T_{\mathsf{fresh}}\}, \sigma_i\}$). π_i including a fresh timestamp T_{fresh}, which proves that D_i is currently neither software compromised nor physically attacked. Provided these new proofs, benign devices from different partitions can assess each others trustworthiness and securely communicate.

Tolerating Absence. In order to tolerate short absence periods caused by benign reasons (e.g., system failure followed by reboot), the tolerance interval t_{tol} around T_{hb}, during which a received heartbeat is accepted can be expanded by a short period of time $t_{absence}$. Consequently, the algorithm checkTime that is responsible for checking the freshness of heartbeats becomes as shown in Figure 5.27.

Figure 5.27 : checkTime **on** D_j

```
if p_i = p_j ∧ T_hb < t < T_hb + δ_t + t_tr + t_absence then
    return true
else if p_i = p_j + 1 ∧ T_hb − δ_t − t_absence < t < T_hb
then
    return true
else
    return false
end
```

The extension of the tolerance interval tolerates devices being absent for a short period of time (e.g., enough for reboot after system failure) without being mistakenly counted as physically attacked. Other reasons for benign absence, such as battery depletion and hardware failure, require V's intervention to replace the battery or repair the device.

5.2.7 *Conclusion*

In this section we presented a second scheme for efficient collective attestation that allows efficient and scalable detection of both software and physical attack on very large networks of embedded devices. Our solution mainly targets autonomous networks. However, it can also be extended to allow reporting to a central verifier as discussed in Section 5.2.6. The solution is based on the adversary model and requirements identified in Section 5.1. Unlike the scheme presented in Section 5.1, it has a performance overhead which is constant in the size of the network. This performance gain came at the cost of more strict assumptions regarding security and device mobility. This solution combines local attestation and absence detection with key management in order to enable efficient detection and isolation of devices that are software compromised or physically attacked. It also leverages a roaming protocol to allow device mobility.

5.3 RELATED WORK

In this section we present work from the literature that is directly related to the solutions presented in this chapter. A survey on existing attestation schemes is presented in Chapter 3.

Physical attacks. We distinguish between three different types of physical attacks: (1) *invasive attack* [133] that allow the adversary to extract a device's secrets by directly accessing its internal components; (2) *semi-invasive attacks* [134] that only require device decapsulation, e.g., optical fault injection, laser scanning, thermal imaging; and (3) *non-invasive attacks* [160] that enable extraction of cryptographic secrets during normal operation, e.g., time or power side channel attacks. While invasive and semi-invasive physical attacks require expensive and sophisticated lab equipment, non-invasive attacks may only require low cost electrical engineering tools. The solutions presented in this chapter aim at detecting invasive and semi-invasive physical attacks that require possession of

the target device for a non-negligible amount (hours to weeks [133, 132]). Non-invasive attacks are considered an orthogonal problem.

Absence Detection. Absence detection is a popular topic in the field of Wireless Sensor Networks (WSN). It has been mainly used to detect failure of sensor nodes. In this context, several solutions have been proposed for static [137] and dynamic topologies (e.g., [67, 37]). However, all these schemes are not designed for an adversarial settings. Other WSN solutions utilize absence detection to detect capture of sensor nodes. These solutions also assume that physical attacks require non-negligible time. However, they either are designed for static networks, or provide probabilistic results allowing false negatives.

Attestation & Key Exchange. A lot of prior work investigated combining key exchange with attestation [122, 116]. SAKE [122] allows neighboring sensor nodes to establish symmetric keys with relying on pre-shared secrets, by basing key establishment on the result of attestation. SAKE relies on a software-based attestation scheme that has unrealistic and strong assumptions (see Section 3.4). On the other hand, researchers have proposed extending the key exchange protocol IKEv2 [80] of IPsec [82] with attestation [116]. The goal of such an extension is ensuring the trustworthiness of the devices involved in running an IPsec connection. This extension mainly targets high-end computing devices connected through legacy networks. It is not suitable for autonomous networks of low-end embedded devices.

5.4 CONCLUSION

In this chapter we investigated the problem of physical attacks and presented two collective attestation solutions that enable the detection of both software and physical attacks in large networks of embedded devices. The two presented solutions are based on slightly different assumptions, have different cost, and are applicable in different scenarios. Section 5.1 provided the basis for detection of physical attacks by defining the adversary model, identifying the requirements, and proposing an attestation solution for centralized networks which is secure under the strongest adversary possible. On the other hand, the solution presented in Section 5.2 enables attestation of both autonomous and centralized networks under a weaker adversary that can physically attack less devices. However, it provides better efficiency. Together the solutions enable detection of software and physical attacks in a wide range of applications. In Chapter 6, we investigate the problem of runtime attacks, and provide an attestation solution for autonomous networks that allow detection of such attacks.

6

DETECTION OF RUNTIME ATTACKS

We are increasingly surrounded by interconnected embedded systems, which collect sensitive information, and perform safety critical operations. Most embedded systems perform simple tasks upon reception of a command, in a predefined manner. However, in recent years, embedded systems have been increasingly designed to carry out autonomous collaborative tasks. Networks of autonomous embedded systems (such as, vehicular ad hoc networks, robotic factory workers, search/rescue robots, and drones are already being used for performing urgent, tiresome, and critical tasks with minimal human intervention. For example, drones are (envisioned to be) used for various non-military tasks, such as search and rescue, construction site management, security and surveillance, cargo delivery, and natural disasters prediction and warning. The secure and safe operation of autonomous embedded networks depends heavily on the trustworthiness of the involved devices. Securing such networks requires ensuring the integrity of data exchanged between device, e.g., commands and sensor readings.

Despite all the benefits of the new emerging applications, the high connectivity and autonomy pose crucial security and safety challenges on the underlying embedded devices and their interactions. Such embedded devices have been the target of various attacks from reverse engineering [55, 61] to (large-scale) software attacks [148, 54, 149]. Morever, in addition to malware and physical attacks, embedded devices represent an attractive target for runtime attacks that exploit software vulnerabilities (such as a buffer overflow vulnerability) to subvert the normal execution of a benign program and execute a malicious functionality, e.g., Return Oriented Programming (ROP) attacks [127]. Consequently, it is not always enough to verify the software integrity of embedded networks in order to ensure their correct operation.

The collective attestation solutions presented in Chapter 4 and 5 enable the detection of both malware infestation and physical attacks. However, they are incapable of detecting runtime attacks. Moreover, existing security solutions that target runtime attacks are either unable to provide protection against data-only attacks, i.e., Control-Flow Integrity (CFI) [8], or are inapplicable to complex systems and not scalable to large autonomous networks, i.e., Control-flow Attestation (CFA) [9].

In this chapter, we aim at securing autonomous embedded networks by securing collaborations between devices using CFA. We present the first secure collective attestation solution for large autonomous networks. Our solution is based on reducing the complexity of CFA, hereby enabling it usage to secure interactions in autonomous systems.

Contribution. We investigate the security of large autonomous networks of embedded devices under runtime attacks. We define the threat model for these networks that allows runtime attacks. And we devise the first collective attestation solution that is capable of securing these network. Our solution is based on redesigning CFA in order to allow its efficient applicability to complex systems. It leverages a novel representation of execu-

tion path to reduce the complexity and overhead for verifying an attestation response. In order to demonstrate feasibility, we show how to implement the solution on recent open source flight controller for drones – Pixhawk PX4.[1] Finally, we present extensive performance evaluation based on our implementation, in addition to simulations of the collaboration scenarios involving thousands of devices. Our solution allows securing a collaboration involving $10,000$ devices in order of seconds. It presents a first step towards secure detection of runtime attacks in autonomous embedded networks.

Outline. After providing a brief overview of the solution in Section 6.1, we present our CFA scheme in Section 6.2, and explain details of our solution in Section 6.3. We describe our implementation and present our performance evaluation results in Section 6.4. Security of this solution is examined in Section 6.5, and this chapter concludes in Section 6.6.

Remark. The results presented in this chapter are due to the author of this work and the result of many intensive discussions and collaboration with Raad Bahmani (TU Darmstadt, Germany), Tigist Abera (TU Darmstadt, Germany), Matthias Schunter (Intel Labs, Darmstadt), Ferdinand Brasser (TU Darmstadt, Germany), and Ahmad-Reza Sadeghi (TU Darmstadt, Germany). Parts of this chapter have been published in [10].

6.1 COLLECTIVE ATTESTATION

6.1.1 *Problem Description and System Model*

We consider a dynamic autonomous network $\mathcal{N}$ that is formed of n interconnected homogeneous devices. The network operator O is responsible for initializing each device D_i in a secure environment as well as for the maintenance of $\mathcal{N}$. We do not assume the existence of a central entity responsible for controlling and assessing the trustworthiness of $\mathcal{N}$. $\mathcal{N}$ may not have a routing protocol in place. However, devices in $\mathcal{N}$ should be able to communicate to their direct neighbors [44, 68, 114, 115] Collaboration between devices in the network is performed through the exchange of messages such as: sensor readings, commands, and status information. The mobility of devices can be involuntary, i.e., guided by ambient factors. Therefore, the topology of the network is unpredictable. A collective attestation solution for autonomous networks allows devices in $\mathcal{N}$ to securely interact performing complex collaborative tasks.

6.1.2 *Requirements Analysis*

Objectives. A collective attestation solution for an autonomous network of embedded devices should at least provide the following three properties:

- *Property #1:* Allow detection of remote attacks including malware infestation and runtime attacks.
- *Property #2:* Be efficient and scalable, i.e., have low overhead on both the verifier and the prover.

1 https://dev.px4.io/en/concept/flight_stack

- *Property #3:* Be applicable to large systems with complex software.

The main objective of collective attestation is satisfying property #1 and property #2 in large autonomous networks. Property #1 provides security guarantees under a strong adversary model that is capable of performing runtime attacks. And, property #3 allows supporting applicability to a wide range of applications including autonomous systems.

Adversary Model. We assume an adversary $\mathcal{A}$ capable of modifying the software on any device D_i in $\mathcal{N}$ except the components protected by hardware, e.g., $\mathcal{A}$ cannot modify code stored in ROM. $\mathcal{A}$ may also exploit a software vulnerability (e.g., a buffer overflow), to subvert the control flow of a benign program, stitch together a sequence of existing machine code instruction, and execute an arbitrarily malicious functionality. However, $\mathcal{A}$ is not capable of tampering with the hardware of any device D_i, i.e., physical attacks are ruled out in this context. Therefore, the hardware security architecture of devices in $\mathcal{N}$ is considered to be immune to attacks. $\mathcal{A}$ can eavesdrop on, modify, replay, or drop any message exchanged between devices in $\mathcal{N}$. We assume that the network operator O is trusted. Further, we assume a stealthy adversary that aims at compromising as many devices as possible while evading detection by our solution, hence, we do not consider Denial of Service (DoS) attacks on the attestation protocol which would reveal $\mathcal{A}$'s presence. By compromising a subset of the devices in $\mathcal{N}$, $\mathcal{A}$ aims at manipulating collaborative tasks of $\mathcal{N}$, through generating and sending malicious data to benign devices.

Device Requirements. The design of our solution aims at satisfying the properties described above. However, satisfying these properties require the ability to remotely attest each device in $\mathcal{N}$ in addition to strong isolation. Consequently, the following requirements should be satisfied by every D_i in $\mathcal{N}$:

- *Integrity measurement:* $\mathcal{A}$ should not be able to tamper with the measurement mechanism responsible for measuring the software integrity of any device D_i.
- *Integrity reporting:* $\mathcal{A}$ should not be able to forge the measurement c_i' of D_i's software that is sent to V.
- *Secure storage:* $\mathcal{A}$ should not have access to the cryptographic key(s) that are used in the attestation protocol.
- *Secure Isolation:* $\mathcal{A}$ should not be capable of manipulating one software module through compromising another software module.

If one of these requirements is not achieved, $\mathcal{A}$ can tamper with D_i's interactions without being detected. Security architectures for low-end embedded devices that satisfy these requirements include: TrustLite [84] and ARM TrustZone-M.[2]

Assumptions. Devices in $\mathcal{N}$ are homogeneous, i.e., have the same hardware and software configurations. Homogeneity implies redundancy. A failed or compromised device D_i can be replaced by another device D_j in $\mathcal{N}$. We assume that every device D_i in $\mathcal{N}$ satisfies the above requirements that allow secure remote attestation and provide strong isolation.

2 https://community.arm.com/processors/trustzone-for-armv8-m/b/blog

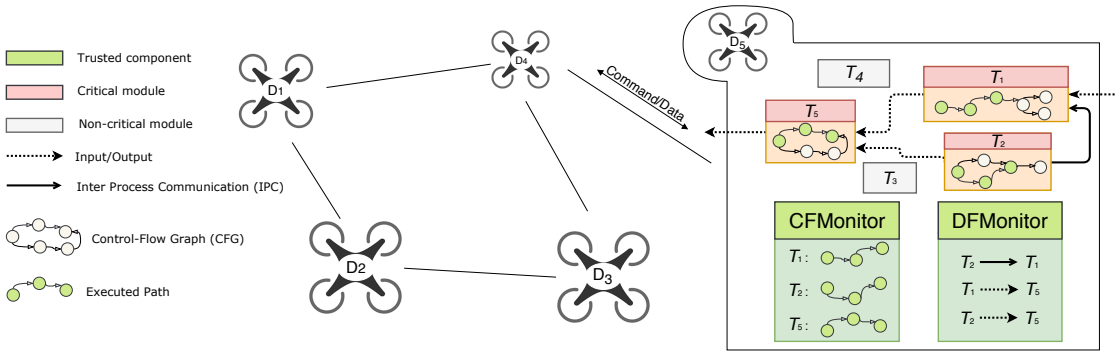


Figure 6.1: Example 5-device network: $D_1, \ldots, D_5$

Further, the sensors and actuators of D_i are assumed to be trusted, i.e., sensors provide correct readings, and actuators behave as instructed. Devices in $\mathcal{N}$ collaborate and communicate with each other, i.e., in order to perform complex tasks every device D_i at least communicates to its neighboring device. Cryptographic primitives are assumed to be secure along with their implementations.

Protocol Overview. The goal of our solution is to enable secure collaboration between individual devices in an autonomous system. In particular, when two devices exchange information, a CFA report, which proves the correct generation and processing of this information is also exchanged between the devices. The main challenge here is allowing efficient CFA that provides data integrity. This challenge is tackled through a novel and efficient execution path representation.

Figure 6.1 shows an overview of our solution in a network formed of five devices D_1 through D_5. While collaborating, devices in $\mathcal{N}$ exchange information (e.g., D_4 and D_5). On each device, a trusted component denoted by DFMonitor, monitors the communication between modules and determines the modules responsible for the generation and processing of the exchanged information. A second trusted components, denoted by CFMonitor, monitors the execution of modules that are identified by DFMonitor to be influencing the exchanged information. CFMonitor creates a CFA report for each of these module based on an efficient execution path representation. The correctness of exchanged information is ensured via a CFA report (generated by CFMonitor) of all modules that participated in its generation and processing (identified by DFMonitor).

6.2 CONTROL-FLOW ATTESTATION

Control-flow Attestation (CFA) allows attesting the exact execution path of an application running on an embedded device (the prover) to a remote verifier. It enables the prover to report the control flow of the program to the verifier, providing assurance on control-flow integrity and detecting runtime attacks. Existing CFA schemes have multiple limitations (see Chapter 3). First, they require binary instrumentation, i.e., rewriting of control-flow instructions (e.g., `jump`, `branch`, etc.) on the prover. Second, they impose a high performance overhead on prover which records every control-flow event. Third,

they impose a significant overhead on the verifier, which is required to store and search a very large database of expected execution paths. The size of this database explodes exponentially with the number of control-flow events. While the first two limitations can be overcome by implementing the prover side of CFA in hardware [49], the third limitation limits the application of CFA to simple software on standalone devices. In fact, applying CFA to autonomous networks where devices with complex software and limited resources should act as both provers and verifiers requires a complete redesign. In this section we present the design of an efficient CFA for autonomous networks that relies on a novel representation of execution paths based on Multiset Hash (MSH) functions [40]. Our CFA scheme represents the logic behind CFMonitor presented in Section 6.1.

6.2.1 *Multiset Hash Function*

A Multiset Hash (MSH) function [40] maps multisets of finite number of unordered members to a fixed length hash value (see Definition 2.8). In the following we present a *multiset-collision resistant* (according to Definition 2.9) MSH function denoted by `MSet-Add-Hash`) from [40]. Details of MSH and multiset-collision resistance are provided in Chapter 2.

`MSet-Add-Hash`. Let $B = \{0,1\}^m$ be the set of all bit strings of length m, and let M be a multiset of elements of B. The number of times a bit string $b \in B$ is in M is called the multiplicity M_b of b in M. Let $H_K : \{0,1\}^{m+1} \to \mathbb{Z}_n^l$ be randomly selected from a pseudo-random family of hash functions. Let $L \approx n^l \approx 2^m$, $L \leqslant n^l$, and $L \leqslant 2^m$. `MSet-Add-Hash` is defined as below.

- *Hash:* Let $r \in B$ be a random nonce. The hash of M is defined as a triple $[h; c; r]$.

$$H(M) = \left[H_K(0,r) + \sum_{b \in B} M_b H_K(1,b) \mod n;\ \sum_{b \in B} M_b \mod L;\ r \right] \Bigg|_{r \in B}$$

- *Extend:* The addition $+_H$ of two hashes $[h; c; r]$ and $[h'; c'; r']$ is:

$$\left[H_K(0,r'') + h - H_K(0,r) + h' - H_K(0,r') \mod n;\ c + c' \mod L;\ r'' \right] \Big|_{r'' \in B}$$

- *Compare:* Two hashes $[h; c; r]$ and $[h'; c'; r']$ are equivalent (i.e., $[h; c; r] =_H [h'; c'; r']$) iff $h - H_K(0,r) = h' - H_K(0,r') \mod n$ and $c = c' \mod L$.

6.2.2 *Efficient CFA*

Before we present the intuition behind our CFA scheme, we briefly describe the most prominent Control-Flow Attestation (CFA) scheme in the literature (i.e., C-FLAT [9]),

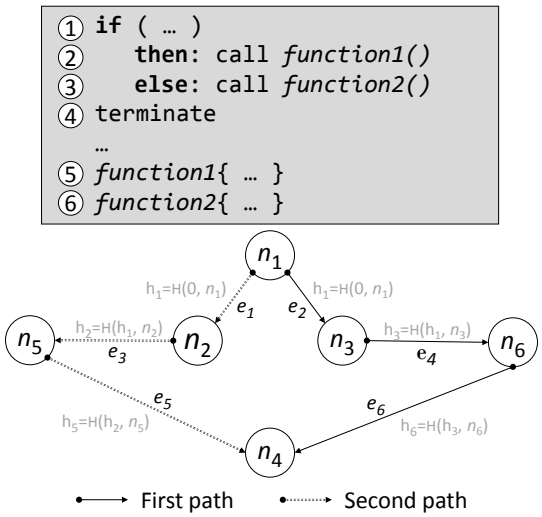


Figure 6.2: Control-Flow Graph (CFG) showing C-FLAT and our CFA scheme

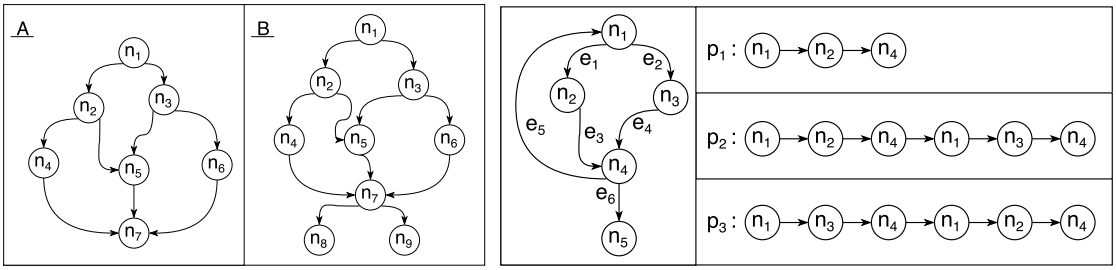
shedding light on its limitations that makes it inapplicable to autonomous systems. For our presentation we utilize the Control-flow Graph (CFG) depicted in Figure 6.2.[3]

In C-FLAT, the binary code to be attested is instrumented with different labels ($n_1, n_2, \ldots$ in Figure 6.2) for each basic block (BBL) ending with any branch instructions (i.e., a node in CFG). When attestation is requested by the verifier, a measurement procedure continuously measures the control-flow path of the executing program. This is done by cumulatively hashing the labels of every executed node, i.e., $h_i = \mathsf{H}(h_{\mathsf{pre}}, n_i)$. For example, consider the execution path of a program to be $n_1 \rightarrow n_2 \rightarrow n_5 \rightarrow n_4$, the attestation response for this execution is $h_4 = \mathsf{H}(\mathsf{H}(\mathsf{H}(\mathsf{H}(0, n_1), n_2), n_5), n_4)$. Note that, this technique provides a fixed-size result as is the case in static attestation. After an attestation response has been generated, it is authenticated along with the nonce received from the verifier using a MAC or a digital signature, and then sent back. Finally, the verifier checks the trustworthiness of the response by searching a database of valid attestation responses (i.e., execution paths).

The number of valid paths, i.e., the size of the verifier's database, depends on the size and complexity of the attested program. Moreover, loops and recursive calls may increase the size of the database indefinitely. The authors propose a solution for the efficient handling of loops and recursion. However, when their complex solution is used, the size of the attestation response may increase indeterminately, and the size of the database can still become very large. Our CFA scheme is based on replacing cumulative hash functions with Multiset Hash (MSH) function that provides a constant-size database as well as a constant-size attestation response. This procedure is not straight forward. It requires special care in order to be secure and efficient.

3 Software code can be presented as a CFG, where nodes represent blocks of code and edges are control-flow transitions (e.g., direct branch).

Obviously, the order of nodes in an execution path is important. Directly using MSH functions over node labels is hereby not secure. The core idea of our CFA scheme is to hash edges in a CFG rather than nodes. An edge can be identified with a label e_i, which is a concatenation of the labels of the two nodes the edge is connecting (e.g., $e_1 = n_1 \| n_2$). For example, the attestation response of the execution path $n_1 \rightarrow n_2 \rightarrow n_5 \rightarrow n_4$ is $H(M = \{e_1 = n_1 \| n_2, e_3 = n_2 \| n_5, e_5 = n_5 \| n_4\})$. It is clear that the attestation response in our CFA scheme has less information than the full execution path. However, the verifier can still (1) detect any added edge to the execution path caused by a Return Oriented Programming (ROP) attack; and (2) construct the set of possible paths a software execution has taken based on the constant-size attestation response, thus, allowing detection of data-only attack that change a program's control flow. In the following a provide a brief analysis of our scheme:



(a) Complexity of our CFA scheme (b) Information loss of our CFA scheme

Figure 6.3: Simple explanatory CFGs

Loops and Break Statements. Loops and break statements constitute a major challenge for C-FLAT as they increase the complexity of the generation and verification of attestation responses. For this reason, C-FLAT is only applicable to very small and simple applications. Our scheme is applicable to more complex programs as it is not based on cumulative hashing of all nodes in a CFG, but rather hashing the multiset of edges in that CFG.

Complexity. Unlike C-FLAT, whose verification complexity is exponential in the number of edges, the verification complexity of our CFA scheme is linear in that number. Consider the two simple CFGs shown in Figure 6.3a. The increase in the number of edges from 9 (in A) to 11 in (in B) increases the number of possible execution paths from 4 to 8, thus, the size of the database increases exponentially. In the contrary, in our scheme only 2 more edges need to be considered on the verifier's end.

Information Loss. Indeed, the attestation response of our scheme carries less information than C-FLAT. However, it provides more information than CFI due to order information preserved inherently in the collected edges themselves. The information loss due to the use of MSH functions depends on the CFG and the executed path. Figure 6.3b shows an example CFG. If path $p_1 = n_1 \rightarrow n_2 \rightarrow n_4$ is executed, then attestation response in our scheme would contain the edges e_1 and e_2. These correspond only to the path p_1. On the other hand, if path $p_2 = n_1 \rightarrow n_2 \rightarrow n_4 \rightarrow n_1 \rightarrow n_3 \rightarrow n_4$ is executed,

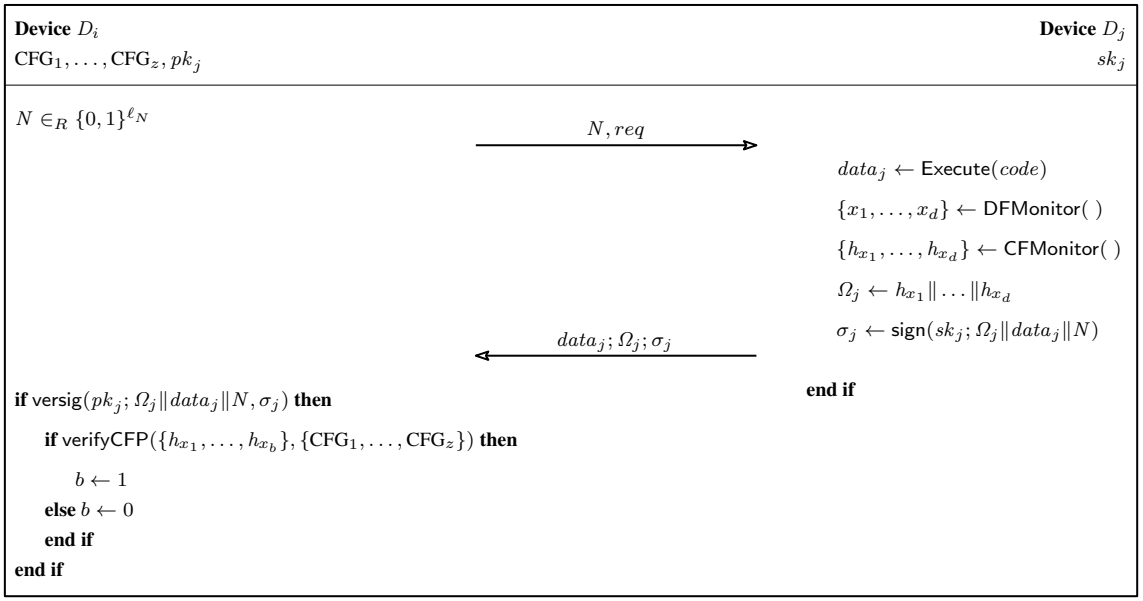


Figure 6.4: Protocol interact

then the attestation response would contain the edges e_1, e_2, e_3, e_4, and e_5. These edges either correspond to p_2 or to $p_3 = n_1 \to n_3 \to n_4 \to n_1 \to n_2 \to n_4$. Consequently, as our scheme preserves the order in which individual edges are executed, it doesn't preserve the order in which different paths inside a loop are executed.

6.3 PROTOCOL DESCRIPTION

Based on the CFA scheme presented earlier, we present a collective attestation solution for autonomous networks that is capable of detecting runtime attacks and securing interaction between collaborating autonomous devices. For the sake of clarity, the solution is first presented as executed between two collaborating devices then later extended for collaborations involving a large number of devices.

6.3.1 *Interaction of Two Devices*

As shown in Figure 6.4, an interaction between two devices D_i and D_j starts by D_i requesting sensitive data $data_j$ from a peer device D_j. In particular, D_i sends D_j a request for data containing a fresh nonce N. Upon receiving a data request, D_j executes the program code necessary for generating the required data, i.e., $data_j \leftarrow \text{Execute}(code)$. During this execution, the communication between software modules is continuously monitored by DFMonitor, which records the IDs $x_1, \ldots, x_d$ of all modules that are capable of changing the value of $data_j$. We denote these modules by *critical modules* with respect to $data_j$. For each critical module identified by DFMonitor, CFMonitor monitors execution and records the control flow of that module according to our Control-Flow Attestation (CFA)

scheme in Section 6.2. At the end of execution, CFMonitor returns a set of attestation responses $h_{x_1}, \ldots, h_{x_d}$ for executed critical modules, i.e., for each module x_i CFMonitor returns the attestation response $h_{x_i} = \{val_{x_i}, M_{x_i}\}$, where $M_{x_i} = \{<e_1,2>, <e_2,1>\}$ means that while executing module x_i, the edge e_1 occurred twice while e_2 occurred once.

The attestation responses $h_{x_1}, \ldots, h_{x_d}$ are sent back to D_i along with the requested data $data_j$. They are authenticated with a digital signature σ_j based on D_j's secret key. Upon receiving the data, D_i verifies its integrity by: (1) verifying σ_j using D_j's public key, and (2) checking the control-flow integrity of all critical modules with respect to $data_j$ using verifyCFP. In particular, verifyCFP verifies that all the edges in a module's attestation response h belong to the CFG of that module. verifyCFP also checks h based on predefined policies, e.g., upper bounds on the number of times loops should be executed. If the verification was successful, D_i gains assurance regarding the integrity of $data_j$ ($b \leftarrow 1$).

6.3.2 *Interaction of Multiple Devices*

In an autonomous network more than two devices can be engaged in a collaboration. The collaboration is secured in this case through the recursive execution of interact (shown in Figure 6.4) between the involved devices. We refer to this protocol as interact+. Through interact+, the exchanged sensitive data is accompanied by the attestation responses of all critical modules *on all collaborating devices.* The main differences between interact and interact+ are outlined below:

- Upon receiving a request from a peer device D_i, D_j generates and sends a new data request to other devices that are involved in the collaboration
- Along with its attestation response, D_j sends D_i the attestation responses its receives from other involved devices.
- If the verification of the attestation responses was successful, D_i concludes that the data was processed correctly by all devices involved in the collaboration.

6.4 IMPLEMENTATION AND EVALUATION

We implemented and evaluated our solutions on top of a popular flight controller for drones – Pixhawk. The underlying Real-Time Operating System (RTOS) used is NuttX, and the adopted autopilot software is PX4. Note that, PX4 is composed of two layers: the flight stack formed of modules responsible for implementing the device's control functionalities, and the middleware responsible for providing basic functionalities to flight stack layer, e.g., communication between modules through an Object Request Broker (uORB) and to the outside world through a Micro Air Vehicle Link (MAVLink).

6.4.1 *Implementation*

We now briefly describe the implementation of the core components of our solution, i.e., DFMonitor and CFMonitor and their integration within PX4. Details of this implementation can be found in [10].

DFMonitor. The goal of DFMonitor is to determine critical modules at runtime and enable their attestation. This is done by (1) extending the MAVLink protocol packets with the necessary fields for attestation requests and responses, and (2) modifying uORB to allow tracking of data exchanged between modules and enable the delivery of fresh data, i.e., through flushing data buffers.

CFMonitor. CFMonitor monitors and records the execution path of modules identified by DFMonitor to be critical. The main components of CFMonitor are a logic that records for every critical module all control-flow events using a Multiset Hash (MSH) value, and code instrumentation that enables the execution of this logic on every integrity-relevant control-flow event.

Integration within PX4. Our solution is integrated within PX4 by modifying the middleware to implement the functionality of DFMonitor, and instrumenting all integrity-relevant control-flow events to call our CFMonitor logic. Further, two other component are implemented that filter unauthorized MAVLink messages (Filter) and authenticate attestation responses (Quoter).

6.4.2 *Evaluation*

We assess the performance of our solution in terms of runtime and present simulations results for collaborations involving up to 10,000 devices. Our performance evaluation is based on the implementation from [10]. The energy consumption of computation on autonomous systems is usually negligible in comparison to other system parts, e.g., actuators. Therefore, we do not present an evaluation of the energy consumption of our solution.

6.4.2.1 *Multiset Hash Function*

Our solution is based on replacing a hash function with a Multiset Hash (MSH) function. In order to understand the direct overhead of this replacement, we compare the runtime of our implementation of the additive MSH function with that of the *Blake2* hash function.[4]

Judging by the runtime of both hash functions in isolation from the overall attestation protocol, it is clear that using a MSH function adds a considerable performance overhead. While hashing one block using a conventional *Blake2* hash functions requires 31 µs, hashing the same block using a MSH function requires 85 µs. This represents a runtime increase of almost 175%. However, using a MSH function has multiple advantages that allow our solution to be better than all existing Control-Flow Attestation

4 https://blake2.net

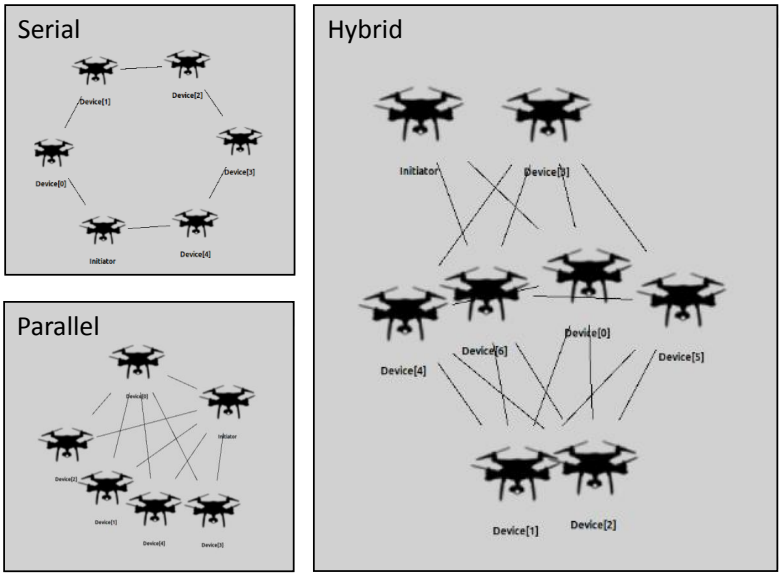


Figure 6.5: Simulated collaboration scenarios (OMNeT++ [104])

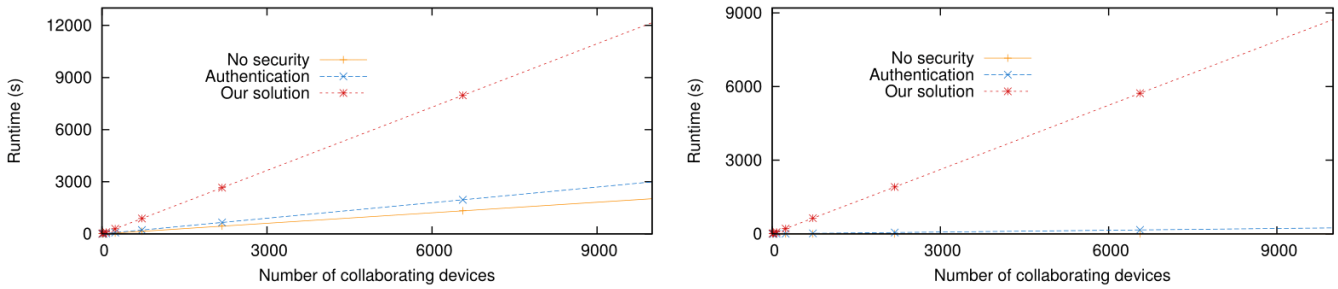
(CFA) schemes [9, 49]. These advantages include: (1) linear verification overhead which enables applicability to complex devices in autonomous systems as opposed to existing schemes that are only applicable to tiny standalone devices, (2) reduction of the communication overhead compared to the naive approach of sending the entire execution path to the verifier, and (3) reduction of required secure rewritable memory compared to the naive approach of calculating the hash of a multiset of edges at the end of program execution using a conventional hash function. Note that, the last approach does not allow performing attestation in parallel to program execution using dedicated hardware [49].

6.4.2.2 *Attestation*

We measured the time required for the generation and verification of attestation reports for various data generation processes. For example, when GPS coordinates are generated on Pixhawk, 13 software modules are typically being executed. Only one of these modules is critical for the GPS data. This represents an improvement of at least 95% over existing CFA schemes, which require attesting all executing modules. The case of GPS data represents one of the extremities of our performance evaluation. Performance gain may vary depending on the data in concern. For example, for the generation of gyroscope sensor data only 2 modules are critical which represents 25% of the 8 executing modules.

6.4.3 *Network Simulation*

We used the OMNeT++ [104] network simulator to assess the performance of our solution for collaborations involving a large number of devices. Our solution was implemented on the application layer, where cryptographic operations were emulated with



(a) Serial collaboration scenario

(b) Parallel collaboration scenario

Figure 6.6: Performance of our solution in serial and parallel collaboration scenarios

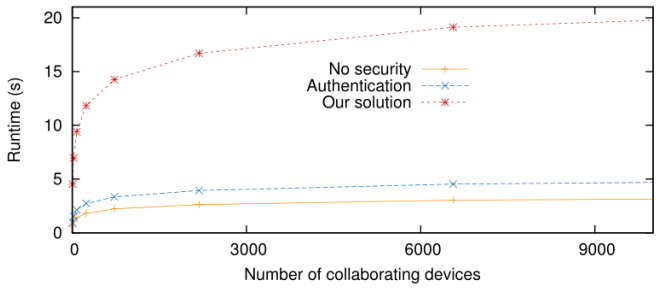


Figure 6.7: Performance of our solution in hybrid collaboration scenario

delays corresponding to real measurements of their execution time on a Pixhawk. For our simulations, the average communication rate between devices was set to 250 Kbps, which corresponds to the effective bandwidth provided by NodeMCU – a WiFi module that can be used for connecting autonomous drones.[5] As shown in Figure 6.5 we considered three popular scenarios: a serial collaboration scenario, where collaboration is executed sequentially on devices; a parallel collaboration scenario, where collaboration is executed in parallel on all devices; and a hybrid collaboration scenario, where collaboration is executed both sequentially and in parallel. While serial and parallel collaboration scenarios represent two extreme cases, the hybrid collaboration scenario represents a realistic use case scenario that falls in between. Further, we simulated various collaboration sizes, which ranged from 10 to 10,000 devices.

The simulated collaboration required each device to generate GPS data and send it to a collaborating peer device. We also considered three different security measures that can be deployed. In particular, we simulated a control case where GPS data is not secured by any means; a basic transmission integrity case where messages are authenticated with a digital signature (ECDSA); and our solution where the integrity of data is ensured by attesting the execution of critical modules. Figure 6.6, and 6.7 show the results of our simulations.

In the following, we briefly analyze the results for each collaboration scenario:

5 `http://nodemcu.com/index_en.html`

Serial Collaboration Scenario. In this scenario the runtime overhead for collaboration, under the three security measures, increases linearly with the collaboration size. The main reason behind this is that involved devices process, authenticate, and exchange data sequentially.

Parallel Collaboration Scenario. In this scenario the runtime overhead for collaboration in the control case is constant in the collaboration size. This is due to the parallel processing and exchange of data. For the two remaining cases, the runtime overhead for collaboration increases linearly with the collaboration size. The main reason behind this is the sequential verification of signatures and attestation responses at the end of the collaboration.

Hybrid Collaboration Scenario. In this scenario the runtime overhead for collaboration, under the three security measures, is logarithmic in the collaboration size. The main reason behind this is the even distribution of the processing, authentication, and attestation burden across all collaborating devices.

Conclusion. Figure 6.7 shows that the overhead of our solution is four times higher than that of basic authentication for the hybrid collaboration scenario. However, the security guarantees provided by our solution are much stronger than those provided by basic integrity protection using authentication. Moreover, unlike existing Control-flow Attestation (CFA) schemes [9, 49], our solution is applicable to complex autonomous systems as it allows efficient verification of attestation responses. Note that, the runtime of our solution can be as low as 1.5 s in a collaboration involving more than 6000 devices.

6.5 SECURITY ANALYSIS

The goal of our collective attestation solution is to enable every device D_i in the network $\mathcal{N}$ to securely interact with every other peer device D_j., i.e., D_i should accept the data coming from D_j and return $b = 1$ if this data is correctly generated by the software on D_j. We formalize this goal as a security experiment $\mathbf{Exp}_{\mathcal{A}}$, where the adversary $\mathcal{A}$ can interact with every device in $\mathcal{N}$. $\mathcal{A}$ has full control over the communication channel between every two devices in $\mathcal{N}$, i.e., it can eavesdrop on, modify, drop, and inject arbitrary messages to any $D_i \in \mathcal{N}$. At the end of the experiment, D_i outputs the result b indicating whether it has accepted maliciously generated data from D_j, following a polynomial number (in ℓ_N, ℓ_h, and ℓ_{sign}) of steps performed by $\mathcal{A}$. The output of D_i represents the result of this security experiment, i.e., $\mathbf{Exp}_{\mathcal{A}} = b$. In the following we provide the definition of secure collective attestation under the pre-described adversary model:

Definition 6.1 (Secure collective attestation under runtime attacks)**.** *Let* f *be a polynomial function in* ℓ_N, ℓ_h, *and* ℓ_{sign}. *We consider a collective attestation scheme to be secure under runtime attacks if the probability* $\Pr\left[b = 1 | \boldsymbol{Exp}_{\mathcal{A}}(1^{\ell}) = b\right]$ *is negligible in* $\ell = f(\ell_N, \ell_h, \ell_{\mathsf{sign}})$.

Theorem 6.1 (Security of our attestation solution)**.** *The attestation solution presented in this chapter is a secure collective attestation solution (Definition 6.1) if the underlying signature scheme is selective forgery resistant, and the Multiset Hash (MSH) function is multiset collision resistant.*

Proof sketch of Theorem 6.1. D_i considers received data from D_j (i.e., $data_j$) as correct and returns $b = 1$ only if the following statements are all correct: (1) The signature σ_j generated over $data_j$, $h_{x_1}, \ldots, h_{x_d}$, and N verifies successfully, where $h_{x_1}, \ldots, h_{x_d}$ represent the set of attestation responses of the critical modules that were involved in the generation of $data_j$, and N is the random nonce sent by D_i; and (2) all the edges in every attestation response h_{x_l} belong to the CFG of the module x_l, and do not violate any of the verification policies. Consider the four cases where the $\mathcal{A}$ attacks: (1) the protocol between D_i and D_j, (2) the communication between software modules on D_j, (3) the binary code of software on D_j, or (4) the control flow of software modules on D_j. It is easy to see that all possible attacks on our solution are covered by a combination of these cases.

We first consider the case where $\mathcal{A}$ attacks the protocol between D_i and D_j. A protocol adversary $\mathcal{A}$ may undermine the security of our solution by: replaying a previously recorded signature over old data along with $data_j$ or forging a signature σ_j over maliciously generated output. However, since the response includes the fresh nonce N, the probability of D_i accepting replayed responses and outputting $b = 1$ is negligible in ℓ_N. Similarly, since the signature scheme is unforgeable, the probability of finding a correct signature is negligible in ℓ_{sign}.

We now consider the case where $\mathcal{A}$ attacks the communication between software modules on D_j. In order to hide the traces of maliciously generated data, $\mathcal{A}$ may try to prevent CFMonitor from recording the execution of one (or more) malicious critical modules, or use a malicious non-critical module to influence the value of data. However, this is not possible since all modules on D_j are strongly isolated and their communication is monitored through DFMonitor which is protected by hardware.

Next we consider the case where $\mathcal{A}$ maliciously modifies the software of D_j in order to undermine the security of our solution. $\mathcal{A}$ may try to: extract the authentication key sk_j used for authenticating attestation responses or modify the software or instrumentation of critical modules in order to maliciously modify the data without being detected. However, these attacks are not possible because the underlying security architecture protects the secrecy of sk_j and the integrity of the software binary.

Finally, we consider the case where $\mathcal{A}$ exploits runtime attacks to enable D_j to generate malicious data. $\mathcal{A}$ may undermine the security of our solution by: injecting malicious code and corrupting control-flow information to redirect execution to the injected code (code injection); exploiting code reuse attacks and maliciously combining code snippets (Return-Oriented Programming) or functions (function reuse attacks); or manipulating variables that affect the program's control flow and execute a non-expected path (non-control-data attacks). However, these attacks will be detected at verification time since: code injection is prevented by the Data Execution Prevention (DEP) technology; code-reuse attacks that induce new edges in the execution path will be detected when checking the attestation response against the CFG, while code-reuse attacks that only incorporate valid edges in the CFG are detected when checking the attestation response against verification policies; and non-control-data attacks that involve unexpected control-flow events will also be detected when they violate one of the verification policies.

Therefore, the probability of $\mathcal{A}$ convincing D_i to accept maliciously modified data from D_j and return $b = 1$ is negligible in ℓ_N, ℓ_h, and ℓ_{sign}. □

6.6 CONCLUSION

In this chapter we investigated the problem of runtime attacks and presented a collective attestation solution that enables the detection of runtime attacks in large autonomous networks. The presented solution allows low-end embedded devices to securely collaborate and exchange data within a large autonomous system. The main component of our solution is a novel execution path representation which reduces the overhead of Control-Flow Attestation (CFA) considerably allowing its applicability to complex systems. As our tests and evaluation results show, the solution presented in this chapter is applicable to real-time systems with tight time constraints. In Chapter 7 we investigate the problem of secure and scalable management of large embedded network.

7

SECURE AND SCALABLE MANAGEMENT

A large number of embedded devices are interconnected providing distributed sensing and control for a wide range of applications. This increased connectivity and the services it provides is commonly referred to as the Internet of Things (IoT). Recent forecasts suggest that the number of such devices will reach 50 billions in 2020 [39]. Examples of domains utilizing IoT devices range from small ecosystems, e.g., building automation, to large installations such as industrial control systems. A fundemental requirement in such systems is the ability to manage IoT device [131]. Device management refers to the task of distributing software updates and commands on devices, as well as monitoring of devices' state.

Unfortuntately, existing management solutions and standards for low-end embedded devices are geared towards the single device setting [131, 103]. These solutions are not scalable to large IoT networks. In particular, the overhead of these solutions increases linearly in the size of the network in the absence of trusted intermediaries. Moreover, utilizing existing approaches for secure data aggreation to solve this problem introduces a linear overhead on the entity managing the network [35, 153].

Unlike the rest of this thesis which focuses on securing large networks of embedded devices against malware infestation (Chapter 4), physical attacks (Chapter 5), and runtime attacks (Chapter 6), In this chapter we aim at providing a secure and scalable management solution of these networks, which we find as important as securing these networks against attack. For example, the inability to update the outdated (possibly vulnerable) software of devices in a network or monitor the status of that update provides an attractive attack surface.

A management process involves sending management commands to target devices, e.g., commands to be executed, and monitoring the state of devices by collecting statistics from the network. The goal of device monitoring is to allow capturing the overall state of the network after distributing (one or multiple) commands. In particular, the entity managing the network might be interested in knowing the percentage of devices that successfully executed a software update command. The management solution we present in this chapter allows performing these tasks in a secure and scalable manner. In particular, it ensures low computational, communication, and storage overhead on both the network devices and the entity managing it. This allows a low-end device, e.g., a smart phone, to efficiently manage a network formed of millions of devices.

Contribution. We investigate the problem of management of embedded networks and design the first secure and scalable management solution for these networks. Our solution allows a single low-power entity to manage a very large number of low-end embedded devices. This is enabled by leveraging an untrusted network of intermediaries, such as smart gateways, which are commonly used as a communication infrastructure for low-end IoT devices. The core of our solution is a finite state machine representation

of the management process which provides a domain independent abstraction. This allows expressing complex managements commands with an accurate high-level representation. We denote this abstraction by Management Finite State Machine (M-FSM). Based on M-FSM we construct a secure command distribution protocol which is fully cacheable. It exploits caching capabilities of intermediate nodes to improve the performance of command distribution and provide scalability. This protocol is generic, i.e., it can be plugged into any pull-based message-response protocol allowing devices in the network to "manage themselves". In our solution, devices selectively pull the management commands that they require. We further devise a scalable protocol for monitoring the state of devices in the network. The monitoring protocol is based on a secure data aggregation scheme from [35], which we extend to allow a constant verification overhead. The monitoring protocol leverages aggregation capabilities of untrusted intermediate nodes to provide constant computational and logarithmic communication overhead in the size of the network, thus allowing the management of millions of devices while keeping communication overhead in the network and computational overhead on the managing entity manageable. In order to demonstrate feasibility, we implemented our solution on Riot-OS – an open source operating system for IoT devices. Further, we present extensive performance evaluation based on our implementation, in addition to simulations of the protocols in networks of up to $1,000,000$ devices demonstrating scalability, and showing the low overhead on the entity managing the network.

Outline. In Section 7.1 we introduce our system model and present the Management Finite State Machine (M-FSM). In Section 7.2 we present our secure data aggregation scheme. We present the details of our command distribution and monitoring protocols in Section 7.3, and describe our implementation in Section 7.4. Performance evaluation of the protocols is then presented in Section 7.5 and their security is examined in Section 7.6. We overview work related to our solution in Section 7.7, and conclude this chapter in Section 7.8.

Remark. The results presented in this chapter are due to the author of this work and the result of many intensive discussions and collaboration with Moreno Ambrosin (Intel Labs, Oregon), Matthias Schunter (Intel Labs, Darmstadt), Mauro Conti (University of Padua, Italy), and Ahmad-Reza Sadeghi (TU Darmstadt, Germany). Parts of this chapter have been published in [13].

7.1 SCALABLE MANAGEMENT

7.1.1 *Problem Description and System Model*

As in Section 4.1.1, we consider a centrally managed dynamic network $\mathcal{N}$ that is formed of a large number n of heterogeneous devices. Manager M is the owner/operator of $\mathcal{N}$, and is the entity responsible for performing the management of every device in $\mathcal{N}$.

An example network is shown in Figure 7.1. We consider four logical entities participating in the protocols: *Manager* (M), *managee* (v_j), *aggregator* (a_k), and *cache* (c_l). A managee v_j is the entity that is managed by M, i.e., its receives commands and reports

its status to M. An aggregator a_k is the entity that is responsible for aggregating status reports coming from managees. And, a cache is the entity that caches management commands coming from M. While managees clearly represent IoT devices, examples of aggregators and caches include gateway and edge devices. In fact, each physical device D_i in $\mathcal{N}$ may act as more than one of these three logical entities. For example, device D_1 in Figure 7.1 acts as managee v_1, aggregator a_1, and cache c_1. Note that, he main goal of aggregators and caches is reducing the communication overhead and speeding up the command distribution and status monitoring processes respectively.

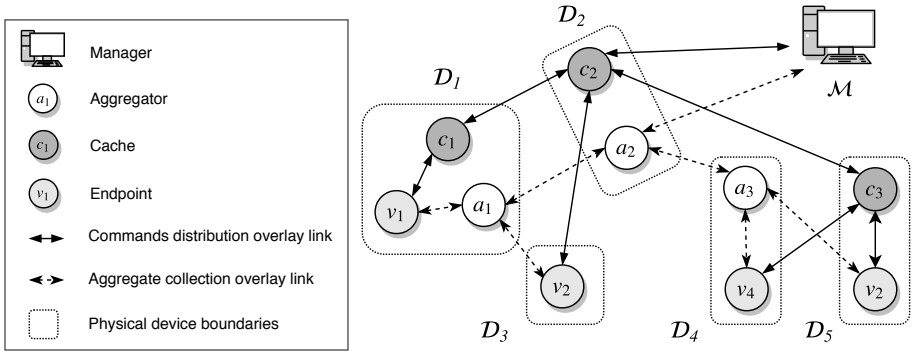


Figure 7.1: System model as a network of devices; each device acts as at least one of the following entities: endpoint (v_j), aggregator (a_l), and cache (c_k)

The three entities form two types of logical trees in $\mathcal{N}$: a *distribution tree* that is formed of caches as inner nodes and managees as leaf nodes, and an *aggregation tree* where inner nodes are aggregators and leaf nodes are managees. Connection between any two entities is purely logical, i.e., it may map to a real network communication (e.g., link between v_2 and a_1) or present an internal communication within one physical device (e.g., link between v_1 and a_1). This generic model allows representing multiple use cases that range from Wireless Sensor Networks (WSN) where every sensor node acts as the three logical entities, to smart environments where IoT devices are managees and gateways act as relay entities (i.e., aggregators and/or caches).

In this model, a secure and scalable management solution enables a manager M to distribute management commands and collect statistics over device status in a secure and efficient manner. The basic idea is that an adversary is not capable of tampering with or forging commands issued by manager or statistics collected from devices in the network.

7.1.2 *Requirements Analysis*

Objectives. The main goal of our solution is to allow secure and scalable management of a very large number of embedded device through a low-end manager M. This translates to reducing computational and communication overhead on the devices in $\mathcal{N}$ and on M, while preserving the authenticity and integrity of distributed commands and collected statistics. Consequently, a secure and scalable management solution for large networks of embedded devices should provide the following properties:

- *Property #1:* Guarantee that management commands are efficiently delivered to all managees in $\mathcal{N}$. We refer to this property as the *outbound efficiency* property.
- *Property #2:* Allow M to efficiently collect statistics regarding the state of managees, e.g., the number of managees that successfully performed a software update. This property is denoted by the *inbound efficiency* property.
- *Property #3:* Guarantee that managees only execute legitimate commands issued by M. We denote this property by the *outbound security* property
- *Property #4:* Ensure the integrity of statistics collected from managees. This property is referred to as the *inbound security* property.

While the satisfaction of property #1 and property #3 allow secure and scalable distribution of management commands, property #2 and property #4 enable secure and scalable collection of device status. Property #1 is based on the Management Finite State Machine (M-FSM) abstraction. And, property #2 and property #4 are achieved through our secure data aggregation scheme.

Adversary Model. As in Section 4.1 we assume $\mathcal{A}$ can eavesdrop on, modify, drop, or replay any message exchanged between all devices in $\mathcal{N}$ and between any device D_i and the manager M. We assume that M is trusted in distributing legitimate management commands. We consider two kinds of attacks: software attacks whereby an adversary can modify all the software on any device D_i in $\mathcal{N}$ except what is protected by hardware; and, physical attacks that allow an adversary to tamper with the hardware of devices, modify their software and extract their secrets that are protected by hardware. We assume $\mathcal{A}$ is capable of mounting both software and physical attacks on all devices that do not act as managees, i.e., relay entities (aggregators and caches) and not trusted. These entities are assumed to be under full control of $\mathcal{A}$. However, $\mathcal{A}$ is incapable of launching physical attacks on devices acting as managees. These devices can only be the target of software attacks. Finally, we assume a stealthy adversary that aims at tampering with the management protocols while evading detection. Hence, we do not consider Denial of Service (DoS) attacks on the management protocols which would reveal $\mathcal{A}$'s presence.

Device Requirements. The design of our solution aims as satisfying all properties #1 to #4. However, in order to satisfy these properties, every device acting as a managee should be equipped with a lightweight security architecture, e.g., SMART [52] and TrustLite [84]. Managees that do not satisfy these requirements may lie about their status possibly affecting the outcome of the statistics collection protocol.

Assumptions. Devices can have different hardware and/or software. However, we assume that every managee (i.e., every physical device D_i acting as managee) in $\mathcal{N}$ is equipped with a lightweight security architecture. We assume that all devices can communicate to the manager M. Finally, we assume that all cryptographic primitives, such as Message Authentication Code (MAC), and their implementations are secure.

7.1.3 *Management Finite State Machine*

The core component of our management solution is a finite state machine representation of the management process, which allows decoupling the realization of this process from any domain specific requirements. This representation provides a domain independent abstraction that allows defining a simple and scalable command distribution process between managees and the manager *M*. The advantage of this representation is that it enables *M* to provide upon request a number of static contents. The static nature of these contents allows their efficient delivery to managees through a distribution tree of caches.

The specification of the management process in our solution is represented by an extended finite state machine – the Management Finite State Machine (M-FSM). M-FSM comprises a number of smaller state machines denoted by sub-M-FSM, each corresponding to the execution of a single command. A simple sub-M-FSM is shown in Figure 7.2. It constitutes of:

- **States:** Every sub-M-FSM comprises at least three different states that can be assumed by a device: (1) the *starting* state which is assumed by devices while waiting for commands; (2) the *attempted execution* state assumed by devices after executing a command; and (3) one or more *terminal* states, e.g., fatal errors. Note that, states are identified by their unique IDs SID.

- **Transitions:** Every sub-M-FSM comprises at least two different transition: A transition between the starting and attempted execution state, that is labeled by the `execute` event. This transition specifies by its corresponding `COMMAND` the command to be executed. And, one or more transitions to the terminal state. Note that, the execution of commands is done by the Execute function, which may write values to global variables. For example, the execution of `COMMAND` writes its output into the variable `out`. Any transition that is starting at the attempted execution state is labeled by a `switch` event on the value of `out`. This transition specifies by its corresponding `OTHER_ACTION` the action to be executed. Further, it may end at a terminal state or a starting state of a different sub-M-FSM.

A graphical representation of a simple sub-M-FSM is shown in Figure 7.2. In this representation states are represented by ovals and transitions are represented by arrows. On top of each arrow we show the events and commands/actions corresponding to the transition. These are separated by the symbol "|". Moreover, the boolean guards which indicate the transition to be chosen are shown in squared bracket. Recall that a sub-M-FSM represents the execution of one command. The execution of complex management processes are created by combining multiple sub-M-FSMs. This allows representing multiple consecutive commands, where executing one command relies on the successful execution of previous commands. Combining two sub-M-FSMs is achieved by adding a new transition from the attempted execution state of one sub-M-FSM to the starting state of the other. This transition is labeled by a `switch` event on the value of `out`.

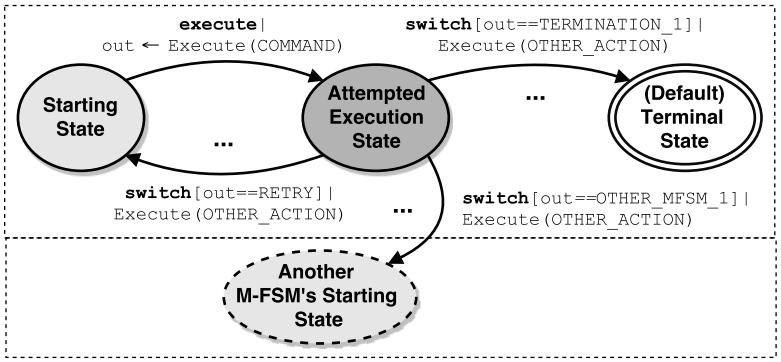


Figure 7.2: A simple sub-M-FSM with 3 states and 4 transitions. One of the transitions is to the starting state of a different sub-M-FSM

Note that, this composability of sub-M-FSMs allows to continuously update an M-FSM by adding new sub-M-FSMs over time. This is particularly important in management scenarios, where the management process cannot be fully defined in advance, e.g., software update. Consequently, at any period of time t_i, the whole management process can viewed by a managee as the command to execute at t_i. This allows providing a management solution having a constant overhead on the managees.

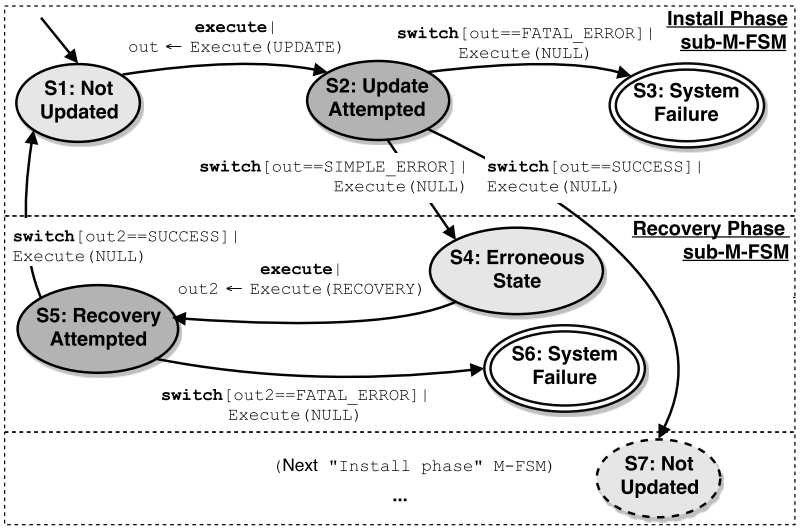


Figure 7.3: Example: Software update management

Software Update M-FSM. As a use case example, we explain a simplified software update management process. The M-FSM for this process is shown in Figure 7.3. Every update process comprises two phases: an installation phase and a recovery phase. Figure 7.3 shows for each phase the corresponding sub-M-FSM.

The starting state of the update process is the "Not Updated" state (S1). From S1 a transition labeled as `execute` allows the device to move to "Update Attempted" state

(S2) when executing the corresponding command `UPDATE` via the Execute function. The output of this execution is then written to the global variable `out`. Based on the value of the variable `out`, the device moves to the next state following one of the transitions labeled as `switch` while executing the action `NULL`. In particular, if the value of `out` is `FATAL_ERROR` the device assumes the terminal "System Failure" state (S3). If the update was successful returning `out` == `SUCCESS`, the device is brought to the starting state of the subsequent sub-M-FSM, i.e., a "Not Updated" state with a different value for SID.

Finally, if update encountered a simple error returning `out` == `SIMPLE_ERROR`, the device moves to the starting state "Erroneous State" (S4) of the second sub-M-FSM corresponding to the recovery phase. From S4 a transition labeled as `execute` allows the device to move to "Recovery Attempted" state (S5) when executing the corresponding command `RECOVERY` through Execute, i.e., recovering to old software. Execute writes its output to the global variable `out2`. Based on the value of `out2`, the device either assume a terminal state "System Failure" (S6) or moves back to the initial "Not Updated" state.

To prevent recovery from being attempted for an infinite number of times. The `RECOVERY` action may keep a counter indicating the number of recovery attempts. When this counter exceeds a certain threshold, the execution of `RECOVERY` returns a fatal error, i.e., `out2` == `FATAL_ERROR`, bringing the device into the terminal state "System Failure" (S7) . Additionally, the transition from state S2 to state S7 may be replaced by a transition to the "Not Updated" state (S1) having a new SID. This would allow avoiding state explosion [146].

7.2 SECURE DATA AGGREGATION

Secure data aggregation enables the reduction of communication overhead while preserving the security of aggregated data (see Section 5.3). In this section we present our integrity preserving data aggregation scheme which we utilize in our monitoring protocol [35]. Our scheme is based on a data aggregation scheme from the Wiresless Sensor Network (WSN) literature [35]. However, unlike the existing scheme, our scheme has a constant verification overhead. Thus, it satisfies the requirements of a secure and scalable management solution. The proposed aggregation scheme enables a querying entity, e.g., the control station of the WSN, to securely and efficiently collect statistics from devices in a large network, e.g., sensor nodes. It comprises two main phases: a collection phase, and a checking phase.

Collection Phase. The querying entity initiates the collection phase by sending a query to all devices in the network. The query indicates the kind of data the querying entity is interested in. It is disseminated throughout the network across a tree of aggregating nodes. The devices in the network located at leaf nodes respond to the query with the required data. Aggregating nodes then recursively aggregate all data coming from their child nodes and send the aggregation results to their parents. Along with the aggregation result, each aggregating node sends to its parent a hash of all the responses it has aggregated, which serves as a commitment to its aggregation. The final aggregated response of all devices in the network is then sent by the root of the tree to the querying

entity. Note that, all hashes generated by all aggregators are forwarded across the tree along with the aggregation results and are also sent back to the querying entity.

Checking Phase. Upon receiving the final response and the commitments, the querying entity initiates the verification phase in order to verify the integrity of the response. The querying entity broadcasts the commitments and the response to all devices in the network. Every device is expected to verify that its contribution has been successfully added to the final response. If the verification was successful, each device replies with a protocol specific acknowledgment. Acknowledgments are authenticated based on multisignatures, which are aggregated along the tree en route to the querying entity. Finally, if the multisignature verifies successfully, the querying entity concludes that the integrity of the aggregated response was preserved.

7.3 PROTOCOL DESCRIPTION

Our solution constitutes of two protocols that are executed between the manager M and the devices D_i in the network $\mathcal{N}$: a command distribution protocol, and a monitoring (or statistics collection) protocol. In this section we describe the details of those protocols.

7.3.1 *Command Distribution Protocol*

The command distribution protocol represents a scalable protocol that allows the distribution of management commands issued by M to every managee v_j in $\mathcal{N}$. The protocol is based on the specification of the Management Finite State Machine (M-FSM). It leverages the abstraction provided by the M-FSM to enable domain independence, minimal memory overhead on managees, and efficient caching that is convenient for Content Delivery Networks (CDNs). For the sake of clarity, we present our command distribution protocol as executed between M and a single managee v_j through a single cache c_l.

The intuition behind the protocol is that every managee v_j in a specific states of M-FSM requests from M the transitions that correspond to this state. Based on these transitions, v_j can move throughout the M-FSM executing management commands and assuming new states. In particular, the transition requested by v_j may either be labeled as `execute` indicating a command `COMMAND` to be executed, or as `switch` indicating a `OTHER_ACTION` action to be performed. The managee v_j requests the transition by sending M (through cache c_l) a management request *req*. The manager M then replies with a management response *resp*. The intermediate caches may cache management responses before forwarding them to the managees in order to reduce the latency of future requests. Caching of management responses is of particular importance in the case of large payloads, e.g., software update [11]. Note that, the assumed communication model for our command distribution protocol is already supported by numerous existing protocols such as CoAP [27]. It is also compatible with recently proposed data centric protocols, e.g., Named-Data Networking (NDN) [78].

Protocol Details. The details of the protocol are shown in Figure 7.4. A managee v_j in state SID sends M a management request demanding a transition from its current state.

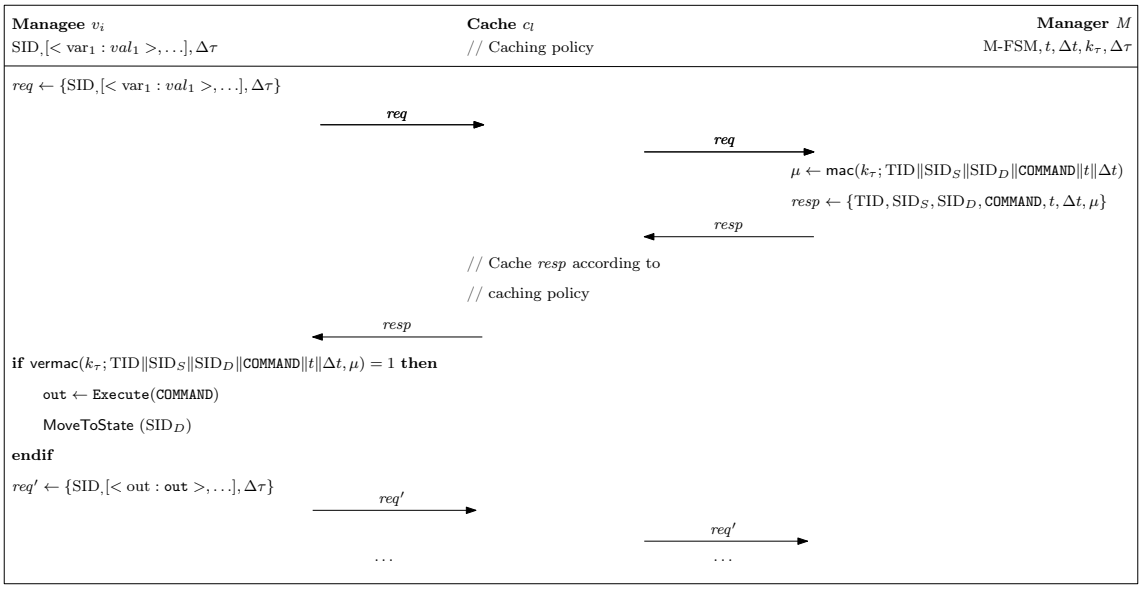


Figure 7.4: Command distribution protocol based on μTesla

The request *req* is formed of the ID SID of v_j's current state, and a set $[< \text{var}_1 : val_1 >, \ldots]$ of M-FSM variables with their respective values. Note that, the transport protocol indicates the appropriate way for including these parameters in *req*. *M* uses these parameters to identify the best management response it should send to v_j. This response *resp* is composed of a transition from the current state SID along with its corresponding management command. Upon receiving *resp*, v_j executes the command via the function Execute and assumes the corresponding attempted execution state indicated by the transition.

After executing the command, v_j sends *M* a new management request containing the output of this execution. As a response, v_j receives a new transition to a (terminal or starting) state – MoveToState.

Note that, if the command has a large payload, e.g., software update, *M*'s response would only contain a pointer to the payload, e.g., a hash. This pointer can be then used to retrieve the payload.

In order to ensure freshness of cached responses, a validity interval Δt and timestamp t are added to each *resp* issued by *M*. To determine whether a response *resp* is fresh, a managee simply checks whether *resp*'s t is within Δt. To ensure availability, caches should allow managees to enforce management requests to be directly served by *M*. This can be done in data centric protocols, such as Named-Data Networking (NDN) [78], through special flags, e.g., `Freshness` and `MustBeFresh`. The application level protocol CoAP [27], on the other hand, provides a `Max-Age` option which indicates the time in seconds after which a response is no longer considered fresh.

Protocol Security. Our command distribution protocol can be combined with different security primitives that allow authenticated broadcast. For example, management responses may be authenticated by *M* with either μTesla [110] or digital signatures. Note that, using μTesla enables applicability to low-end devices, which are not capable of per-

forming computationally extensive public key operations, while preserving cacheability and public verifiability of the distributed management commands.

Based on the chosen authentication primitive, M's responses are either sent to managees with a MAC or a digital signature. Authenticating commands with digital signatures is fairly simple. Each response is signed by M using its secret key sk_M, and verified by managees using M's public key pk_M.

Command distribution based on μTesla is shown in Figure 7.4. When μTesla is used, each response is authenticated with a MAC based on the symmetric key k_τ, which is valid for a short interval of time τ. This key k_τ is disclosed at time $\tau + d$ allowing managees to verify the MACs on management responses authenticated with that key. In other words, at time τ, the managee v_j downloads and buffers a management response that it requires. Then, at time $\tau + d$, i.e., when k_τ is disclosed, v_j verifies the response and executes the corresponding command.

Building a key sequence in μTesla is based on hash chains, i.e., the key k_τ for interval τ is generated as the hash of the key $k_{\tau+1}$ for interval $\tau + 1$. Since different applications may demand different key disclosure times, M maintains multiple key sequences that have different key disclosure times. When M receives a management request *req*, it authenticates the response *resp* based on the key from sequence indicated by *req*.

Note that, management responses authenticated with digital signatures can be cached permanently. However, responses on μTesla expire when the authentication key is disclosed. Managees can freely select, through *req*, whether the required management response should be authenticated with a MAC or a digital signature. This allows managees to choose between performing extensive computation and waiting for authentication key to be disclosed. There are various factors that may alter a managee's choice on this matter, .e.g., energy source, computational power, or application specific time bounds. Additionally, a managee may choose between different key sequences, with different disclosure times, based on multiple factors including its degree of synchronization. This allows managees to tradeoff security level for a delay in receiving the management response. Finally, we consider the number of different key sequences maintained by M and their key disclosure times to be design parameters that are application specific.

7.3.2 *Statistics Collection protocols*

The monitoring protocol represents a scalable protocol that allows the manager M to collect statistics on the status of devices in $\mathcal{N}$, e.g., the number of managees that assume a specific state in the Management Finite State Machine (M-FSM). The protocol is based on the secure data aggregation scheme presented in Section 7.2. Thus, it allows an aggregate value to be generated within the network, while imposing a verification overhead which constant in networks size. As in Chapter 4 and 5, the entities performing the aggregation are not trusted. They form an aggregation tree that has M as a root and managees as leaf nodes.

Protocol Details. The details of the protocol are as follows:

- M initiates the first phase protocol by broadcasting the ID SID of a state of interest, e.g., "Not Updated" state (S7) in Figure 7.3. M's request can be authenticated with either a digital signature or a MAC based on μTesla.
- Every managee v_j replies to its parent aggregator with a bit indicating whether it currently assumes the state with ID SID, i.e., 1 if v_j is in SID and 0 otherwise.
- Every aggregator a_k sums all the values coming from its child nodes, and forwards the result to its parent. Eventually, M receives a final aggregate value indicating the total number of managees in a state wit h ID SID.
- M initiates the second phase of the protocol by broadcasting the received results. This broadcast is also authenticated as above, i.e., through digital signatures or μTesla.
- Upon receiving the second broadcast, every managee v_j checks whether its contribution was correctly integrated in the final aggregate value (see Section 7.2). If the check was successful, v_j replies with a multisignature on a protocol specific "ACK" message. Otherwise, v_j sends back a "NACK" message to its parent node.
- Aggregators aggregate the multisignatures en route to M according to the definition of a multisignature scheme in Chapter 2. Eventually, M receives a single multisignature for the whole network $\mathcal{N}$.
- Having all public keys of managees in $\mathcal{N}$, M can verify the received multisignature and determine whether the total number of managees in SID was generated correctly.

If the verification of the multisignature failed, M concludes that an error has occurred. This error might be benign, e.g., caused by a connection problem. The error might also mean that aggregators were trying to maliciously modify the outcome of the protocol.

Device Inspection. In some scenarios, it might be desirable to detect the IDs of devices in a specific state SID rather than their mere number (as described above). In this case, M may issue a command requesting the IDs of all managees in that state. As a response, every managee in SID will report back to M with its ID. Note that, the overhead of this procedure is linear in the size of the network, and it can easily overwhelm the whole network unless constrained to a small number of devices.

7.4 IMPLEMENTATION

We implemented the two main entities involved in our solution, i.e., the managee v_j and the manager M. Our implementation of the managee targets IETF Class 1 and 2 devices [26]. It was realized as a Client Agent Module for Riot OS [18, 64], which executed both the command distribution and monitoring protocols.[1] On the other hand,

1 Our implementation of the monitoring protocol is based on the multisignature from [24] which uses bilinear pairings [25].

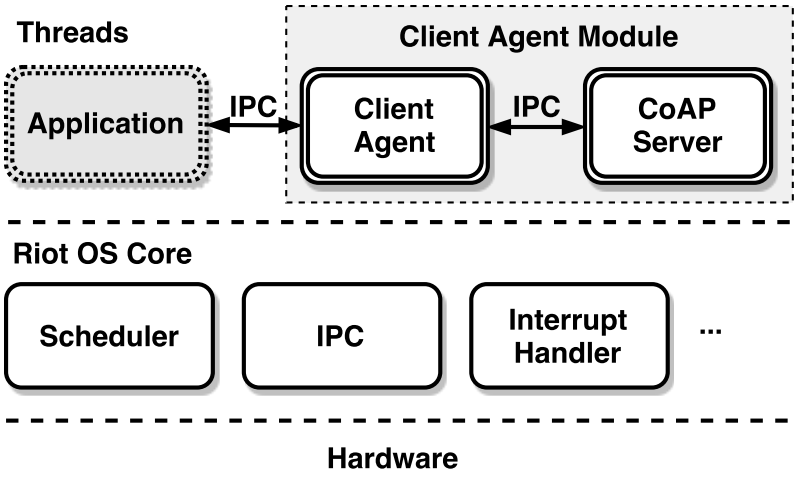


Figure 7.5: Client Agent Module of managees for Riot-OS

the implementation of M comprises a simple application with several exposed APIs for command distribution. M's application also collects periodic statistics regarding the status of managees. The implementation of M will not be presented in this section.

Riot OS [18, 64] is an open source operating system for low-end embedded devices. It constitutes a basic microkernel, in addition to a set of modules that can be included if necessary for the execution of an application. On Riot OS there is no differentiation between processes and threads. Every application executes on its own thread. It can create a large number of other threads, which is limited by the size of available memory. The Riot module we implemented for managees exposes a number of APIs that can by used by any application is order to allow the command distribution protocol. This module utilizes CoAP [27] transfer protocol for the distribution of management commands from M and for the collection of statistics from v_j.

Our implementation is shown in Figure 7.5. The Client Agent Module is formed of two components: a Client Agent and a simple CoAP Server. These two components execute on different threads and interact through Inter Process Communication (IPC). The module receives commands from the network and forwards them through IPC to applications that are performing management process. The communication between the managee and the manager exploits a set of minimal CoAP REST APIs. Note that, this implementation can make use of an edge node translating CoAP requests into HTTP [50].

Let `SERVER_IP` denote the IP address of the aggregator closest to the managee, a command request through CoAP is:

$$\texttt{coap}:\texttt{//[SERVER\_IP]/sid?sid} = \texttt{SID\&...},$$

Note that $\texttt{sid} = \texttt{SID}$ represents the obligatory part of the request. Similarly, a statistics collection request through CoAP for the state with ID SID is:

$$\texttt{coap}:\texttt{//[BROADCAST\_IP]/assess/?nonce} = \texttt{N\&sid} = \texttt{SID}.$$

7.5 PERFORMANCE EVALUATION

We assess performance of our solution in terms of communication, memory overhead, and runtime. We further present simulations results for networks of up to $1,000,000$ devices. Our performance evaluation is based on the implementation in Section 7.4. The managees in our evaluation constitute low-end embedded devices with specifications comparable to those of M3 Open Node devices from the SensLAB testbed [2]. They are equipped with 64 KBytes of RAM, a 32 bits ARM Cortex M3 processor running at 72 MHz, and a 2.4 GHz transceiver [3]. On the other hand, the manager is comparable to a Raspberry Pi Mod B. It features 512 MBytes of RAM, 2 GBytes of flash memory, and a 32 bits ARM Cortex A processor running at 700 MHz.

Note that, we utilized the embedded system library in [145] for the implementation of our multisignature scheme, and the mbedTLS library [1] for implementation of the remaining cryptographic primitives, i.e., MAC and ECDSA.

Memory Cost. Every managee v_j in $\mathcal{N}$ should store the following: (1) a key pair (sk_i, pk_i) for the multisignature scheme; (2) one M-FSM state which constitutes of the state ID SID; and (3) the public key pk_M of the manager M (in case the protocol security is based on digital signatures) or the root of a hash chain (in case the protocol security is based on μTesla). The memory cost of D_i is around 322 Bytes in the case of digital signatures, and 310 Bytes in the case of μTesla. Low-end embedded devices, which we target in this solution, have more than $1,024$ Bytes of Flash memory. Our solution uses 31.4% of this memory when based on digital signatures, and 30.3% based on μTesla. Finally, aggregators do not store any data, and the storage overhead of a cache is dependent on the size of cached data and overall capacity of the cache's memory.

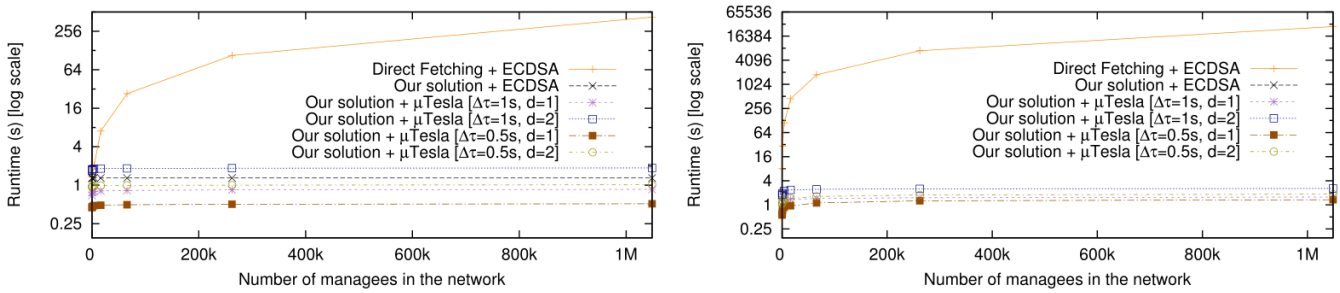
Communication Cost. We used HMAC based on SHA-1 as our MAC implementation, and ECDSA as our digital signature scheme, i.e., $\ell_{\mathsf{mac}} = 160$ and $\ell_{\mathsf{sign}} = 320$. We further used a 32 bit timestamp and chose $\ell_{id} = 32$. As a consequence, MACs are 20 Bytes each. The variables t, Δt, SID, and TID are 4 Bytes each. And, digital signatures are 40 Bytes each. Note that, using μTesla adds an overhead of broadcasting a key every interval τ, i.e., 30 Bytes, which we do not include in our analysis. For the command distribution protocol, every managee v_j has a communication overhead, which is between 80 and 334 Bytes when digital signatures are used, and 37 and 291 Bytes when μTesla is used. On the other hand, the statistics collection protocol is formed of two phases. In the first phase, every managee v_j receives 24 Bytes, and sends 26 Bytes. The communication overhead of the second phases is logarithmic in the network size. It depends on the number of managees and the depth of the aggregation tree. This overhead is caused by the size of data required by each device to check whether its contribution was integrated correctly into the final aggregate value [110]. More precisely, every managee has a communication overhead which is upper bounded by receiving $26 \times (d + s)$ Bytes and sending 84 Bytes, where s is the number of managees in $\mathcal{N}$ and d is the depth of the aggregation tree. For example, if $\mathcal{N}$ is formed of $n = 2^{10}$ devices, the depth of the aggregation tree is $d = 14$, and the number of managees is $s = 2^4 = 16$, then every managee would receive 780 Bytes.

Table 7.1: Performance of cryptographic functions

Function	M3 node from SenLab Run-time (ms)	Raspberry Pi Mod B Run-time (ms)
$H(m) \in \mathbb{G}_1$ (1)	360.319	89.168
$g_1^x,\ g_1 \in \mathbb{G}_1$	494.619	124.604
$g_1 \times g_1',\ g_1, g_1 \in \mathbb{G}_1$	23.615	8.459
$e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$	– (2)	1.736
hash(3)	0.102	0.031
hmac(3)	0.408	0.124
verECDSA (3)	1181.140	– (2)

(1) Hash computed on a 20 Bytes nonce
(2) Not performed by the device during the protocol
(3) For 64 Bytes

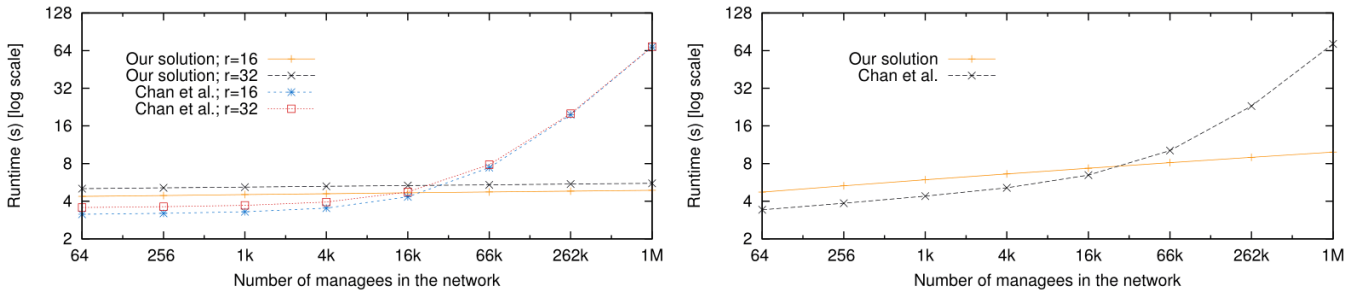
Runtime. The runtime of the two protocols presented in this chapter is dominated by communication overhead and the time required to perform cryptographic operations. We measured the runtime overhead of the cryptographic primitives adopted in our protocols on both the managees, i.e., M3 device, and the aggregator/manager, i.e., Raspberry Pi Mod B. The results are shown in Table 7.1.



(a) First scanrio (b) Second scenario

Figure 7.6: Performance of command distribution

Simulation Results. We used the OMNeT++ [104] network simulator to assess the performance of our solution in large scale. The protocols were implemented on the application layer, where cryptographic operation were emulated with delays corresponding to real measurements of their execution time on M3 node and Raspberry Pi Mod B. For our simulations, the average communication rate between managees and aggregators was set to 75 Kbps, which corresponds to the effective bandwidth for ZigBee [135]. The communication rate among aggregators/caches and between them and the manager was set to 10 Mbps. We simulated two different scenarios. In the first scenarios devices acting as managees are low-end devices, while those acting as aggregators/caches are



(a) First scenario

(b) Second scenario

Figure 7.7: Performance of statistics collection

high-end devices. In the second scenario, devices acting as managees, aggregators, and caches are all low-end devices.

For the first scenario, we simulated various network sizes, which ranged from 2^6 to $2^{20} = 1,048,5761$. The aggregation/caching tree formed a binary tree rooted at M. The number of devices in this tree was set to be proportional to the number of managees, where the ratio r between aggregators/caches and managees is constant. In this scenario, the topology of the network is assumed to be static, e.g., infrastructure in smart environments. For the second scenario, we also varied the size of the network from 2^6 to $2^{20} = 1,048,5761$, and we assumed that the managees formed a binary tree rooted at M. Figure 7.6, and 7.7 show the results of our simulations (in the two scenarios) for command distributions and statistics collection protocols respectively.

For evaluating the command distribution protocol in the first scenario we set r to 32. Further, we configured the caches to utilize a First-In-First-Out (FIFO) caching policy. Every managee requests, at a random time from 0 s to 1 s, a command from M. We simulated the two cases where commands are authenticated based on ECDSA and on μTesla. In the case of μTesla, managees can only verify and execute a command upon the disclosure of the authentication key at $\tau + d$, where $\Delta\tau \in \{0.5, 1\}$ s and $d \in \{1, 2\}$ s. We compared our command distribution protocol to direct fetching from manager. Figure 7.6 shows the time needed by a managee to fetch a command from M as function of number of managees for all simulated cases. Our command distribution protocol performs significantly better than direct fetching. Further, the time needed to fetch a command is logarithmic in the number of managees. The figure also shows that when $d = 1$ μTesla performs better than digital signatures. However, this performance improvement comes at the cost of stricter assumptions and a more complex key management procedure.

Our experiments highlights the scalability of the command distribution protocol, which leads to maximum cacheability of commands. The presented results are compatible with prior evaluation results, e.g., evaluation results of Updaticator [11], which performed a small scale evaluation over a Named-Data Networking (NDN) network [78].

Finally, for evaluating the command distribution protocol in the first scenario, we considered the cases where $r = 16$ and $r = 32$. We compared our statistics collection protocol to the data aggregation scheme from [35]. Figure 7.7 shows the time needed by

the manager to collect statistics regarding a state in M-FSM as function of number of managees. The runtime of our statistics collection protocol is logarithmic in the number of managees compared to the linear runtime imposed by the data aggregation scheme from [35]. Note that, the overhead of both protocols is mainly caused by the verification of the acknowledgment in the second phase, which is constant in our statistics collection protocol and linear in [35]. This overhead is also dependent on the height of the aggregation tree, which in turn depends on the ratio r. Our statistics collection protocol performs better when $r = 32$ than $r = 16$. Recall that, this is caused by the amount of data that should be exchanged to allow a device to verify its contribution.

When the number of managees is small, e.g., less than $32,768$ entities for $r = 16$, the aggregation scheme from [35] performs better than our statistics collection protocol (e.g., 4 s compared to 4.4 s). This is due to our adoption of multisignatures which are far more costly for low-end devices than a simple MAC. However, as the number of managees increases, the runtime overhead of the data aggregation scheme grows rapidly compared to the slowly growing runtime of our statistics collection protocol. As a consequence, when the number of managees exceeds a certain threshold, our statistics collection protocol starts performing better than the data aggregation scheme (e.g., 4.7 s compared to 5.4 s). This result proposes devising a hybrid approach, where the manager chooses between MACs and multisignatures based on the number of managees in $\mathcal{N}$.

7.6 SECURITY ANALYSIS

The goal of our management service is to enable the manager M to securely (1) issue commands to every managee v_j in the network $\mathcal{N}$ (command distribution protocol), i.e., v_j should only correctly execute commands that are issued by M; and (2) collect statistics concerning the status of managees in $\mathcal{N}$ (statistics collection protocol), i.e., M should only accept correct aggregated statistics that reflect that actual state of managees in $\mathcal{N}$. This security goal can be formalized as a security experiment $\mathbf{Exp}_{\mathcal{A}}$, where the adversary $\mathcal{A}$ can interact with every device in $\mathcal{N}$ as well as M. $\mathcal{A}$ has full control over the communication channel between every two devices in $\mathcal{N}$, and between $\mathcal{N}$ and M, i.e., it can eavesdrop on, modify, drop, and inject arbitrary messages to any $D_i \in \mathcal{N}$ and M. $\mathcal{A}$ can maliciously modify the software of any device in $\mathcal{N}$. It may also exploit physical proximity to tamper with the hardware of any device that does not act as a managee. At the end of the experiment, v_j outputs its decision b_1 indicating whether it has accepted a malicious command and M outputs its decision b_2 indicating whether it has accepted a manipulated aggregate value, following a polynomial number (in ℓ_{hash}, ℓ_{mac}, ℓ_N, and ℓ_{sign}) of steps performed by $\mathcal{A}$. The OR of the two outputs represents the result of this security experiment, i.e., $\mathbf{Exp}_{\mathcal{A}} = b \mid b = b_1 \vee b_2$. In the following we provide the definition of secure management under the pre-described adversary model:

Definition 7.1 (Secure management service)**.** *Let f be a polynomial function in ℓ_{hash}, ℓ_{mac}, ℓ_N, and ℓ_{sign}. A management service is secure if the probability* $\Pr\left[b = 1 | \boldsymbol{Exp}_{\mathcal{A}}(1^{\ell}) = b\right]$ *is negligible in* $\ell = f(\ell_{\mathsf{hash}}, \ell_{\mathsf{mac}}, \ell_N, \ell_{\mathsf{sign}})$.

Theorem 7.1 (Security of our management solution)**.** *The management solution presented in this chapter is a secure management service (Definition 7.1) if the underlying MAC, digital signature, and multisignature schemes are selective forgery resistant, and* µ*Tesla is secure.*

Proof sketch of Theorem 7.1. We consider the two cases where the $\mathcal{A}$ either tricks v_j to return its decision $b_1 = 1$ or M to return $b_2 = 1$:

- $\mathcal{A}$ *attacks command distribution:* As a response to a management request containing Δt, the managee v_j receives from the manager M a management command $\{TID, ..., t, \Delta t\}$ authenticated with either a digital signature or a MAC. v_j accepts and returns $b_1 = 1$ only if the verification of the signature (resp. MAC) was successful. In order to convince v_j to return $b_1 = 1$, $\mathcal{A}$ then must forge a digital signature (resp. MAC). However, since the signature (resp. MAC) scheme is selective forgery resistant, the probability of $\mathcal{A}$ forging it is negligible in ℓ_{sign} (resp. ℓ_{mac}). Moreover, since the keys used for creating MACs in the command distribution protocol are based on a hash chain, where $k_{\tau-1} \leftarrow hash(k_\tau)$. $\mathcal{A}$ could use an older key $k_{\tau-1}$ in order to generate the MAC or extract the current key k_τ. However, since µTesla is secure (i.e., the hash used for creating the key sequence is collision resistant and preimage resistant), the probability of generating a correct MAC or extracting the current key is negligible in ℓ_{hash}.

- $\mathcal{A}$ *attacks statistics collection:* In order to undermine the security of the statistics collection protocol and convince manager to accept a manipulated aggregate value and return $b_2 = 1$, $\mathcal{A}$ may try to: (1) modify the query sent by manager (i.e., state of interest) during the collection phase, or (2) forge or replay a multisignature over an acknowledgment of a managee v_j during the verification phase. Attacking the collection phase requires $\mathcal{A}$ to either forge a digital signature or a MAC based on µTesla. However, this attack is negligible in ℓ_{sign}, ℓ_{mac}, and ℓ_{hash}. Similarly, since the multisignature is selective forgery resistant, attacking the verification phase is negligible in ℓ_{sign} and ℓ_N.

Therefore, the probability of $\mathcal{A}$ convincing a managee v_j to execute a malicious command, or the manager M to accept manipulated statistics is negligible in ℓ_{hash}, ℓ_{mac}, ℓ_N, and ℓ_{sign}. □

7.7 RELATED WORK

In this section we present work from the literature that is directly related to the solution presented in this chapter.

Device Management The Open Mobile Alliance (OMA) recently proposed The Lightweight Machine to Machine protocol (LWM2M) [103], which aims at allowing secure management of embedded devices. However, this solution is geared toward the single-device setting. It allows the management of individual devices only, and is not applicable to large scale scenarios. In princple, prior work on management of embedded devices either focuses on managing the IoT network itself [120], or requires managing each device

individually [131]. We consider existing work to be complementary to the management solution that we present in this chapter. In particular, our solution could leverage existing work to allow inspection of individual devices, performing one time management operations, or maintaining the network topology when required. Ambrosin et al. presented a software update protocol that exploits cache-enabled networks to allows secure and scalable delivery of confidential updates [11]. The solution presented by the authors is based on a Named-Data Networking (NDN) network. Further, it does not allow the manager to collect statistics regarding the state of managees in the network. Burke et al. proposed a security architecture for instrumented environments [29]. Their solution is also based on NDN networks. It allows the exchange of authentic and confidential messages (i.e., commands and acknowledgments) between the manager and managees. However, the proposed security architecture does not allow messages to be multicasted. Therefore, it is also not applicable for managing devices in large scale scenarios.

Secure data aggregation. Secure data aggregation allows reducing the communication overhead in WSN, by combining data of different sensor, while maintaining its integrity and/or secrecy. A lot of prior work has been done on secure aggregation schemes that preserve data integrity. Proposed schemes rely on witness-based solutions [51], cryptographic techniques [72, 93, 34, 33, 156, 87, 106], or trust relations [105]. Further, they either have high communication and computational overhead [51, 100, 36], require globally shared keys [93], or involve asymmetric cryptography [87], which is computationally expensive. Note that, while existing secure aggregation aim at reducing the computational and communication overhead on low-end sensor nodes, they consider the base station to be a powerful entity capable of performing a large number of operation in order to verify the aggregate value. In this context, node congestion, i.e., the upper bound on the communication overhead of one node, is considered as an important metric for evaluating secure data aggregation schemes. Chan et al. presented a secure data aggregation scheme for WSN, which has an overall node congestion of $\mathcal{O}(\Delta \log^2 n)$ [35]. This value is further reduced by Frikken et al. to $\mathcal{O}(\Delta \log n)$ through a new commitment scheme [58]. However, the two schemes require the powerful base station to create and XOR MACs generated by all sensor nodes. Such schemes are not applicable in our model, were a low-power entity should be capable of securely collecting statistics from a very large number of devices.

7.8 CONCLUSION

In this chapter we investigated the problem of secure and scalable management of embedded devices and presented a secure management solution that is applicable in large scale. The presented solution allows a low-power manager to securely distribute command and monitor the status of a very large number of devices. Our command distribution protocol relies on a management process abstraction based on finite state machines. This abstraction allows to decouple the management process from any domain specific parameters, thus, enabling efficient retrieval of management commands. On the other hand, our device monitoring protocol is based on a secure data aggregation scheme

which has a verification overhead that is logarithmic in the number of managed entities. As our evaluation results show, the solution presented in this chapter is extremely scalable and has a manageable runtime overhead.

8

DISCUSSION AND CONCLUSION

Large networks of embedded systems are becoming increasingly popular. These networks represent an attractive target for attack as they are often connected to the legacy internet, they collect and process sensitive information through increasingly deployed sensors, and they perform safety-critical physical actions through various types of actuators. A large number of attacks have been launched on embedded networks, e.g., Stuxnet [148] and HVAC [149] attacks.

As presented in this dissertation, remote attestation aims at establishing trust in remote devices by allowing a trusted verifier to check the software integrity of a remote device. While existing remote attestation protocols enable the detection of malware infestation attacks on a standalone device, they fall short in securing emerging networks of embedded devices. These protocols are not only inefficient in large scale, they are also insecure under any realistic adversary model for large embedded networks.

In this dissertation we aim at securing large embedded systems against a wide range of possible attacks. We present the design and implementation of multiple collective attestation solutions that are capable of detecting malware infestation attacks, physical attacks, and runtime attacks on devices in embedded networks. Further, since device management and software update represents an important starting point for an attack on embedded devices, and since existing secure management schemes are not applicable in large scale, we present the design and implementation of a secure and scalable management solution for large embedded networks. The main results of this dissertation are summarized in Section 8.1 followed by a brief discussion on future research directions in Section 8.2.

8.1 DISSERTATION SUMMARY

Detection of Malware Infestation. In Chapter 4 we investigate malware infestation problem in the context of large embedded networks. We further present two collective attestation solutions that allow the detection of malware infestation attacks on multiple devices in the network. The two solutions have different properties, they are based on different assumption and cryptographic primitives, and they provide different security guarantees in different application scenarios. They provide a tradeoff between efficiency and security. Further, we present a systematic treatment of collective attestation that allows establishing such a security service on solid ground. Our treatment enables the analysis and comparison among various solutions, as well as the construction of a generic collective attestation solution that completes the aforementioned tradeoff by providing stronger security guarantees at the cost of minimal additional requirements and runtime overhead.

Detection of Physical Attacks. In Chapter 5 we investigate the problem of physical attacks in the context of large embedded networks. We again present two collective attestation solutions that allow the detection of both malware infestation and physical attacks on multiple devices in the network. The two solutions have different overall cost, are based on different assumptions, and are applicable in different scenarios. We start by providing the basis for the detection of physical attacks through defining the adversary model and identifying the requirements for a secure collective attestation solution under physical attacks. We then provide solutions that are applicable for both centralized and autonomous networks under different adversarial capabilities. The two solutions presented in Chapter 5 allow detection of malware infestation and physical attacks for a wide range of application scenarios.

Detection of Runtime Attacks. In Chapter 6 we investigated the problem of runtime attacks in the context of large embedded networks. We present a collective attestation solution that allows the detection of runtime attacks in large autonomous networks. This solution enables low-end embedded devices to securely collaborate and exchange data within the autonomous system. It is based on three major building blocks: data integrity attestation that allows the integrity of sensitive data to be defined based on the integrity of the software that process this data; modular attestation that decomposes large software into small interacting modules, thus reducing the overhead of attestation; and execution path representation based on Multiset Hash (MSH) functions which reduces the overhead of attestation considerably.

Secure and Scalable Management. In Chapter 7 we investigated the problem of secure device management in the context of large embedded networks. We present a secure and scalable management solution that is capable of managing and collecting statistics over large number of devices. This solution allows a resource-constrained manager to distribute management commands and monitor the status of a large embedded network in a secure and efficient manner. The command distribution part of our solution is based on abstracting the management process by means of finite state machines. Abstraction allows decoupling management from all domain specific parameter, hereby enabling secure and efficient distribution of management commands. Statistics collection, on the other hand, relies on a our secure data aggregation scheme that has a logarithmic verification overhead in the size of the network.

8.2 FUTURE RESEARCH DIRECTIONS

Privacy Preserving Attestation. All proposed attestation protocols expect the prover to send the verifier its current software configuration in plaintext. Similarly, in the collective attestation solutions presented in this dissertation, devices in an embedded network either send their software configuration to all neighbors when they first join the network, or to the verifier during the execution of the attestation protocol. Consequently, an adversary could exploit attestation to learn the software configuration of a device, which can then be used to find and exploit a known software vulnerability for that particular software configuration. This problem is even more critical for collective attestation

that allows the adversary to scale its attack for a large number of devices. Naive solutions to this problem, e.g., encryption of the attestation response or randomization of the measurement process, impose a high verification overhead and do not scale to large embedded networks. It might also be desirable in certain scenarios to hide the software configuration of the prover from the authentic verifier while allowing it to verify its trustworthiness. This is also not possible with the current attestation paradigm, where the verifier needs to have a priori knowledge of the prover's software configuration. The consequences of this open access to a device's software configuration caused by attestation should be investigated. This would allow determining the severity of the problem on both single-device and collective attestation. It would also enable devising privacy preserving attestation protocols that do not reveal a device's software configuration to an external eavesdropper or to an authentic verifier.

Detection of Advanced Runtime Attacks. Control-Flow Attestation (CFA) allows the detection of some classes of runtime attacks that are not prevented by any other mitigation technique, e.g., Control-Flow Integrity (CFI), as it allows the prover to report its exact execution path to the verifier. However, in order to detect such attacks, the verifier should be able to inspect the received execution path and determine illegal control-flow events. It is usually assumed that the verifier has a priori knowledge regarding certain characteristics of the execution path, which we refer to in this dissertation as the verification policy, e.g., an upper bound on the number of iterations of a loop, or privileged vs. non-privileged paths. Regardless of the validity of such an assumption, it does not allow the detection of all classes of runtime attacks, e.g., attacks that do not violate any known verification policy. CFA should devise means for constructing a comprehensive set of verification policies that allow the verifier to detect a wide range of runtime attacks. The verification process should be automatically generated from the context of execution. Possible directions for solving this problem might include leveraging machine learning techniques to classify execution paths based on the context of execution, or utilizing static and dynamic analysis for a better understanding of programs' behavior. Finally, CFA should also allow the detection of runtime attacks that do not change a program's execution path by measuring and reporting the data flow within this program.

Post Quantum Attestation. Current and emerging computing paradigms and architectures, e.g., quantum computing, present a formidable threat to existing security services, including attestation, as they threaten the security of their underlying cryptographic primitives. The consequence of quantum computing on the security of attestation should be investigated. This would enable devising single-device and collective attestation solutions that are secure under a powerful quantum adversary.

9

ABOUT THE AUTHOR

Ahmad Ibrahim is a research assistant at the Technische Universität Darmstadt and the Intel Collaborative Research Institute for Collaborative Autonomous & Resilient Systems (Intel CRI-CARS), Germany. In 2013, he received his M.Sc. in Computer Science from Saarland University, Saarbrücken, Germany. His research has mainly focused on securing embedded networks against strong adversaries that are capable of injecting malware, hijacking the control-flow of executing software through memory corruption attacks, and capturing and physically attacking the hardware of embedded devices. To secure these networks, he developed multiple protocols that are based on attestation, absence detection, and control-flow tracking.

AWARDS

- 1st place in M.Sc. of Computer Science, Lebanese University.
- 3rd place in B.Sc. of Computer Science, Lebanese University.
- Scholarship by a Lebanese organization to complete doctoral studies in Germany.
- ACM Student Travel Grant to attend ACM CCS'16
- ACM Student Travel Grant to attend ACM WiSec'17
- ACM and SIGDA Student Travel Grant to attend ICCAD'18
- Participant in the ACM Student Research Competition at ICCAD'18

ACADEMIC ACTIVITIES

Ahmad had the privilege to be involved in a number of academic activities, e.g., assisted Prof. Sadeghi in reviewing papers for major IT-Security conferences (IEEE S&P, USENIX Security, ACM CCS, NDSS), and hardware conferences (DAC, ICCAD).

Member of the student program committee of 40th IEEE Symposium on Security and Privacy (S&P'19)

PEER-REVIEWED PUBLICATIONS

Ivan De Oliveira Nunes, Ghada Dessouky, Ahmad Ibrahim, Norrathep Rattanavipanon, Ahmad-Reza Sadeghi, Gene Tsudik. Toward Systematic Design of Collective Attestation Protocols. In *Proceedings of the 39th IEEE International Conference on Distributed Computing Systems*, ICDCS'19, 2019.

Tigist Abera, Raad Bahmani, Ferdinand Brasser, Ahmad Ibrahim, Ahmad-Reza Sadeghi, Matthias Schunter. DIAT: Data Integrity Attestation for Resilient Collaboration of Autonomous Systems. In *Proceedings of the 26nd Annual Network and Distributed System Security Symposium*, NDSS'19, 2019.

Ahmad Ibrahim, Ahmad-Reza Sadeghi, Gene Tsudik. HEALED: HEaling & Attestation for Low-end Embedded Devices. In *Proceedings of the 23rd International Conference on Financial Cryptography and Data Security*, FC'19, 2019.

Tigist Abera, Ghada Dessouky, Ahmad Ibrahim, Ahmad-Reza Sadeghi. LiteHAX: Lightweight Hardware-Assisted Attestation of Program Execution. In *Proceedings of the 37th IEEE International Conference On Computer Aided Design*, ICCAD'18, 2018.

Ahmad Ibrahim, Ahmad-Reza Sadeghi, Gene Tsudik. US-AID: Unattended Scalable Attestation of IoT Devices. In *Proceedings of the 37th IEEE International Symposium on Reliable Distributed Systems*, SRDS'18, 2018.

Moreno Ambrosin, Mauro Conti, Ahmad Ibrahim, Ahmad-Reza Sadeghi, Matthias Schunter. SCIoT: A Secure and sCalable end-to-end management framework for IoT Devices. In *Proceedings of the 23rd European Symposium on Research in Computer Security*, ESORICS'18, 2018.

Shaza Zeitouni, Ghada Dessouky, Orlando Arias, Dean Sullivan, Ahmad Ibrahim, Yier Jin, Ahmad-Reza Sadeghi. ATRIUM: Runtime Attestation Resilient Under Memory Attacks. In *Proceedings of the 37th IEEE International Conference On Computer Aided Design*, ICCAD'17, 2017.

Ahmad Ibrahim, Ahmad-Reza Sadeghi, Shaza Zeitouni. SeED: Secure Non-Interactive Attestation for Embedded Devices. In *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec'17, 2017.

Moreno Ambrosin, Mauro Conti, Ahmad Ibrahim, Gregory Neven, Ahmad-Reza Sadeghi, Matthias Schunter. SANA: Secure and Scalable Aggregate Network Attestation. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security*, CCS'16, 2016.

Ahmad Ibrahim, Ahmad-Reza Sadeghi, Shaza Zeitouni, Gene Tsudik. DARPA: Device Attestation Resilient to Physical Attacks. In *Proceedings of the 9th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec'16, 2016.

N. Asokan, Ferdinand Brasser, Ahmad Ibrahim, Ahmad-Reza Sadeghi, Matthias Schunter, Gene Tsudik, Christian Wachsmann. SEDA: Scalable Embedded Device Attestation. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, CCS'15, 2015.

SUBMITTED MANUSCRIPTS

Ghada Dessouky, Ahmad Ibrahim, Ahmad-Reza Sadeghi. SoK: The Hitchhiker's Guide to the Attestation. To be submitted to *Proceedings of the 41st IEEE Symposium on Security and Privacy*, S&P'20, 2020.

INVITED PAPERS

Sibin Mohan, Mikael Asplund, Gedare Bloom, Ahmad-Reza Sadeghi, Ahmad Ibrahim, Negin Salajageh, Paul Griffioen, Bruno Sinopoli. Special Session: The Future of IoT Security. In *Embedded Systems Week*, ESWEEK'18, 2018.

EXTENDED ABSTRACTS

Ahmad Ibrahim. Securing Embedded Networks through Secure Collective Attestation. In *Proceedings of the 16th ACM International Conference on Mobile Systems, Applications, and Services*, MobiSys'18, 2018.

Ahmad Ibrahim. Collective Attestation: for a Stronger Security in Embedded Networks In *Proceedings of the 37th IEEE International Symposium on Reliable Distributed Systems*, SRDS'18, 2018.

TECHNICAL REPORTS

N. Asokan, Ferdinand Brasser, Ahmad Ibrahim, Ahmad-Reza Sadeghi, Matthias Schunter, Gene Tsudik, Christian Wachsmann. SEDA: Scalable Embedded Device Attestation. *Technical Report TUD-CS-2015-1195*, 2015.

POSTERS

Moreno Ambrosin, Mauro Conti, Ahmad Ibrahim, Gregory Neven, Ahmad-Reza Sadeghi, Matthias Schunter. POSTER: Toward a Secure and Scalable Attestation. In *Proceedings of the 9th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec'16, 2016.

BIBLIOGRAPHY

[1] https://tls.mbed.org/. ARM mbedTLS cryptographic library.

[2] IoT-LAB: a very large scale open testbed. https://www.iot-lab.info/.

[3] IoT-LAB M3 Open Node. https://www.iot-lab.info/hardware/m3/.

[4] Real-time clocks (rtc) ics. https://www.maximintegrated.com/en/products/digital/real-time-clocks.html. Accessed: 2015-09-30.

[5] Jeep Hacking 101. http://spectrum.ieee.org/cars-that-think/transportation/systems/jeep-hacking-101, 2015.

[6] Amazon EC2 Instance Types. https://aws.amazon.com/ec2/instance-types/, 2016.

[7] N. Ababneh, S. Selvakennedy, and K. Almi'Ani. NBA: A novel broadcasting algorithm for wireless sensor networks. In *IFIP International Conference on Wireless and Optical Communications Networks*, 2008.

[8] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow Integrity Principles, Implementations, and Applications. *ACM Transactions on Information and System Security (TISSEC ' 09)*, 13(1), 2009.

[9] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik. C-flat: Control-flow attestation for embedded systems software. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, 2016.

[10] T. Abera, R. Bahmani, F. Brasser, A. Ibrahim, A.-R. Sadeghi, and M. Schunter. Diat: Data integrity attestation for resilient collaboration of autonomous system. In *26th Annual Network & Distributed System Security Symposium (NDSS)*, NDSS '19, January 2019.

[11] M. Ambrosin, C. Busold, M. Conti, A.-R. Sadeghi, and M. Schunter. Updaticator: Updating Billions of Devices by an Efficient, Scalable and Secure Software Update Distribution over Untrusted Cache-enabled Networks. In *Proceedings of the 19th European Symposium on Research in Computer Security*, ESORICS'14, 2014.

[12] M. Ambrosin, M. Conti, A. Ibrahim, G. Neven, A.-R. Sadeghi, and M. Schunter. SANA: Secure and Scalable Aggregate Network Attestation. In *Proceedings of the 23rd ACM Conference on Computer & Communications Security*, CCS '16, 2016.

[13] M. Ambrosin, M. Conti, A. Ibrahim, A.-R. Sadeghi, and M. Schunter. SCIoT: A Secure and sCalable End-to-End Management Framework for IoT Devices. In

Proceedings of the 23rd European Symposium on Research in Computer Security, ESORICS'18, 2018.

[14] W. Arbaugh, D. Farber, and J. Smith. A secure and reliable bootstrap architecture. In *IEEE Symposium on Security and Privacy*, 1997.

[15] F. Armknecht, A.-R. Sadeghi, S. Schulz, and C. Wachsmann. A security framework for the analysis and design of software attestation. In *ACM Conference on Computer and Communications Security*, 2013.

[16] N. Asokan, F. Brasser, A. Ibrahim, A.-R. Sadeghi, M. Schunter, G. Tsudik, and C. Wachsmann. SEDA: Scalable Embedded Device Attestation. In *Proceedings of the 22nd ACM Conference on Computer & Communications Security*, CCS '15, 2015.

[17] A. M. Azab, P. Ning, and X. Zhang. Sice: A hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, 2011.

[18] E. Baccelli, O. Hahm, M. Gunes, M. Wahlisch, and T. C. Schmidt. Riot os: Towards an os for the internet of things. In *Computer Communications Workshops (INFOCOM WKSHPS), 2013 IEEE Conference on*. IEEE, 2013.

[19] A. Bagherzandi, J.-H. Cheon, and S. Jarecki. Multisignatures secure under the discrete logarithm assumption and a generalized forking lemma. In *Proceedings of the 15th ACM Conference on Computer & Communications Security*, CCS '08, 2008.

[20] K. A. Bailey and S. W. Smith. Trusted virtual containers on demand. In *The ACM Workshop on Scalable Trusted Computing*, 2010.

[21] A. Becher, Z. Benenson, and M. Dornseif. Tampering with motes: Real-world physical attacks on wireless sensor networks. In J. Clark, R. Paige, F. Polack, and P. Brooke, editors, *Security in Pervasive Computing*, volume 3934 of *Lecture Notes in Computer Science*. 2006.

[22] M. Bellare, C. Namprempre, and G. Neven. Unrestricted aggregate signatures. In L. Arge, C. Cachin, T. Jurdzinski, and A. Tarlecki, editors, *International Colloquium on Automata, Languages and Programming, ICALP 2007*, volume 4596 of *Lecture Notes in Computer Science*, 2007.

[23] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM conference on Computer & Communications Security*, CCS '93. ACM, 1993.

[24] A. Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme. In *Proceedings of the 6th International Workshop on Theory and Practice in Public Key Cryptography: Public Key Cryptography*, PKC '03, 2003.

[25] D. Boneh, C. Gentry, B. Lynn, and H. Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In E. Biham, editor, *Advances in Cryptology*, volume 2656 of *EUROCRYPT '03*, 2003.

[26] C. Bormann, M. Ersue, and A. Keranen. Terminology for constrained-node networks. Technical report, 2014. IETF RFC-7228.

[27] C. Bormann and Z. Shelby. Block-Wise Transfers in the Constrained Application Protocol (CoAP). Technical report, 2016. RFC-7959.

[28] F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl. TyTAN: Tiny Trust Anchor for Tiny Devices. In *Proceedings of the 52Nd Annual Design Automation Conference*, 2015.

[29] J. Burke, P. Gasti, N. Nathan, and G. Tsudik. Securing instrumented environments over content-centric networking: the case of lighting control and ndn. In *Proceedings of the 2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2013.

[30] S. A. Camtepe and B. Yener. Key distribution mechanisms for wireless sensor networks: a survey. Technical report, 2005.

[31] X. Carpent, K. ElDefrawy, N. Rattanavipanon, and G. Tsudik. Lightweight swarm attestation: A tale of two lisa-s. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, 2017.

[32] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente. On the difficulty of software-based attestation of embedded devices. In *Proceedings of the 16th ACM Conference on Computer & Communications Security*, CCS '09, 2009.

[33] H. Chan, A. Perrig, B. Przydatek, and D. Song. SIA: Secure information aggregation in sensor networks. *Journal of Computer Security*, 2007.

[34] H. Chan, A. Perrig, and D. Song. Secure hierarchical in-network aggregation in sensor networks. In *ACM Conference on Computer and Communications Security*, 2006.

[35] H. Chan, A. Perrig, and D. Song. Secure hierarchical in-network aggregation in sensor networks. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, 2006.

[36] C.-M. Chen, Y.-H. Lin, Y.-C. Lin, and H.-M. Sun. RCDA: Recoverable concealed data aggregation for data integrity in wireless sensor networks. *IEEE Transactions on Parallel and Distributed Systems*, 2012.

[37] W. Chen, S. Toueg, and M. Aguilera. On the quality of service of failure detectors. *Computers, IEEE Transactions on*, 51(5), 2002.

[38] Y.-G. Choi, J. Kang, and D. Nyang. Proactive code verification protocol in wireless sensor network. In O. Gervasi and M. L. Gavrilova, editors, *Computational Science and Its Applications – ICCSA 2007*, 2007.

[39] Cisco. "forecast on internet of things". `newsroom.cisco.com/feature-content?type=webcontent&articleId=1208342`.

[40] D. Clarke, S. Devadas, M. van Dijk, B. Gassend, and G. E. Suh. *Incremental Multiset Hash Functions and Their Application to Memory Integrity Checking*. 2003.

[41] M. Conti, R. Di Pietro, L. V. Mancini, and A. Mei. Emergent properties: Detection of the node-capture attack in mobile wireless sensor networks. In *Proceedings of the First ACM Conference on Wireless Network Security*, WiSec '08, 2008.

[42] M. Conti, R. D. Pietro, L. V. Mancini, and A. Mei. Mobility and cooperation to thwart node capture attacks in manets. *EURASIP J. Wireless Comm. and Networking*, 2009, 2009.

[43] V. Costan, L. F. G. Sarmenta, M. van Dijk, and S. Devadas. The trusted execution module: Commodity general-purpose trusted computing. In G. Grimaud and F.-X. Standaert, editors, *Smart Card Research and Advanced Applications*, 2008.

[44] E. Şahin. Swarm robotics: From sources of inspiration to domains of application. In *Swarm Robotics*. 2005.

[45] A. Datta, J. Franklin, D. Garg, and D. Kaynar. A logic of secure systems and its application to trusted computing. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, 2009.

[46] L. Davi, A.-R. Sadeghi, and M. Winandy. Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In *The ACM Workshop on Scalable Trusted Computing*, 2009.

[47] G. de Meulenaer, F. Gosset, O.-X. Standaert, and O. Pereira. On the energy cost of communication and cryptography in wireless sensor networks. In *IEEE International Conference on Wireless and Mobile Computing*, 2008.

[48] G. Dessouky, T. Abera, A. Ibrahim, and A.-R. Sadeghi. Litehax: Lightweight hardware-assisted attestation of program execution. In *2018 International Conference On Computer Aided Design (ICCAD'18)*, 2018.

[49] G. Dessouky, S. Zeitouni, T. Nyman, A. Paverd, L. Davi, P. Koeberl, N. Asokan, and A.-R. Sadeghi. Lo-fat: Low-overhead control flow attestation in hardware. In *54th Design Automation Conference (DAC'17)*, 2017.

[50] E. Dijk, A. Rahman, T. Fossati, S. Loreto, and A. Castellani. Internet-Draft: Guidelines for HTTP-CoAP Mapping Implementations. Technical report, 2016. IETF-draft.

[51] W. Du, J. Deng, Y.-S. Han, and P. Varshney. A witness-based approach for data fusion assurance in wsn. In *IEEE Global Telecommunications Conference*, 2003.

[52] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito. SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust. In *Network and Distributed System Security Symposium*, 2012.

[53] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, Dec. 2007.

[54] F-Secure Labs. BLACKENERGY and QUEDAGH: The convergence of crimeware and APT attacks, 2016.

[55] H. Fereidooni, J. Classen, T. Spink, P. Patras, M. Miettinen, A.-R. Sadeghi, M. Hollick, and M. Conti. Breaking fitness records without moving: Reverse engineering and spoofing fitbit. In *Proceedings of the 20th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2017.

[56] A. Francillon, Q. Nguyen, K. B. Rasmussen, and G. Tsudik. Systematic treatment of remote attestation. *IACR Cryptology ePrint Archive*, 2012, 2012.

[57] A. Francillon, Q. Nguyen, K. B. Rasmussen, and G. Tsudik. A minimalist approach to remote attestation. In *Design, Automation & Test in Europe*, 2014.

[58] K. B. Frikken and J. A. Dougherty, IV. An efficient integrity-preserving scheme for hierarchical sensor aggregation. In *Proceedings of the First ACM Conference on Wireless Network Security*, WiSec '08, 2008.

[59] F. Gandino, B. Montrucchio, and M. Rebaudengo. Key management for static wireless sensor networks with node adding. *IEEE Transactions on Industrial Informatics*, 2014.

[60] J. A. Garay and L. Huelsbergen. Software integrity protection using timed executable agents. In *Proceedings of the 2006 ACM Symposium on Information, Computer and Communications Security*, ASIACCS '06, 2006.

[61] L. Garcia, F. Brasser, M. H. Cintuglu, A.-R. Sadeghi, O. Mohammed, and S. A. Zonouz. Hey, My Malware Knows Physics! Attacking PLCs with Physical Model Aware Rootkit. In *24th Annual Network & Distributed System Security Symposium*, NDSS, 2017.

[62] R. Gardner, S. Garera, and A. Rubin. Detecting code alteration by creating a temporary memory bottleneck. *IEEE Transactions on Information Forensics and Security*, 2009.

[63] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, 2003.

[64] O. Hahm, E. Baccelli, H. Petersen, and N. Tsiftes. Operating Systems for Low-End Devices in the Internet of Things: A Survey. *IEEE Internet of Things Journal*, 3(5), 2016.

[65] V. Haldar, D. Chandra, and M. Franz. Semantic remote attestation: A virtual machine directed approach to trusted computing. In *Virtual Machine Research And Technology Symposium*, 2004.

[66] H. Härtig, M. Hohmuth, N. Feske, C. Helmuth, A. Lackorzynski, F. Mehnert, and M. Peter. The nizza secure-system architecture. In *Proceedings of the first EAI International Conference on Collaborative Computing: Networking, Applications and Worksharing*, 2005.

[67] N. Hayashibara, A. Cherif, and T. Katayama. Failure detectors for large-scale distributed systems. In *Reliable Distributed Systems, 2002. Proceedings. 21st IEEE Symposium on*, 2002.

[68] F. Higgins, A. Tomlinson, and K. M. Martin. Threats to the swarm: Security considerations for swarm robotics. *International Journal on Advances in Security*, 2009.

[69] C. Hsin and M. Liu. Self-monitoring of wireless sensor networks. *Comput. Commun.*, 29(4), 2006.

[70] C.-f. Hsin and M. Liu. A distributed monitoring mechanism for wireless sensor networks. In *Proceedings of the 1st ACM Workshop on Wireless Security*, WiSE '02, 2002.

[71] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. "Data-oriented programming: On the expressiveness of non-control data attacks". In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, S&P'16, 2016.

[72] L. Hu and D. Evans. Secure aggregation for wireless networks. In *Symposium on Applications and the Internet Workshops*, 2003.

[73] Y.-C. Hu, A. Perrig, and D. Johnson. Packet leashes: A defense against wormhole attacks in wireless networks. In *IEEE Computer and Communications*, 2003.

[74] A. Ibrahim, A.-R. Sadeghi, and G. Tsudik. Us-aid: Unattended scalable attestation of iot devices. In *Proceedings of the 37th IEEE International Symposium on Reliable Distributed Systems*, SRDS '18, 2018.

[75] A. Ibrahim, A.-R. Sadeghi, and G. Tsudik. HEALED: HEaling & Attestation for Low-end Embedded Devices. In *Proceedings of the 23rd International Conference on Financial Cryptograpy and Data Security*, FC'19, 2019.

[76] A. Ibrahim, A.-R. Sadeghi, G. Tsudik, and S. Zeitouni. DARPA: Device Attestation Resilient against Physical Attacks. In *Proceedings of the 9th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '16, 2016.

[77] A. Ibrahim, A.-R. Sadeghi, and S. Zeitouni. SeED: Secure Non-Interactive Attestation for Embedded Devices. In *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '17, 2017.

[78] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard. Networking named content. In *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '09, 2009.

[79] B. Kauer. Oslo: Improving the security of trusted computing. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, SS'07, 2007.

[80] C. Kaufman. Internet Key Exchange (IKEv2) Protocol. RFC 4306 (Proposed Standard), 2005.

[81] R. Kennell and L. H. Jamieson. Establishing the genuinity of remote computer systems. In *USENIX Security Symposium*, 2003.

[82] S. Kent and K. Seo. Security Architecture for the Internet Protocol. RFC 4301 (Proposed Standard), 2005.

[83] C. Kil, E. Sezer, A. Azab, P. Ning, and X. Zhang. Remote attestation to dynamic system properties: Towards providing complete system integrity evidence. In *The IEEE/IFIP International Conference on Dependable Systems and Networks*, 2009.

[84] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan. TrustLite: A Security Architecture for Tiny Embedded Devices. In *European Conference on Computer Systems*, 2014.

[85] J. Kong, F. Koushanfar, P. K. Pendyala, A.-R. Sadeghi, and C. Wachsmann. PUFatt: Embedded platform attestation based on novel processor-based PUFs. In *Design Automation Conference (DAC)*, 2014.

[86] X. Kovah, C. Kallenberg, C. Weathers, A. Herzog, M. Albin, and J. Butterworth. New results for timing-based attestation. In *IEEE Symposium on Security and Privacy*, 2012.

[87] V. Kumar and S. Madria. Secure hierarchical data aggregation in wireless sensor networks: Performance evaluation and analysis. In *IEEE International Conference on Mobile Data Management*, 2012.

[88] Y. Li, J. M. McCune, and A. Perrig. Sbap: Software-based attestation for peripherals. In A. Acquisti, S. W. Smith, and A.-R. Sadeghi, editors, *Trust and Trustworthy Computing*, 2010.

[89] Y. Li, J. M. McCune, and A. Perrig. VIPER: Verifying the integrity of peripherals' firmware. In *ACM Conference on Computer and Communications Security*, 2011.

[90] J. Liu, Y. Xiao, S. Li, W. Liang, and C. L. P. Chen. Cyber security and privacy issues in smart grids. *IEEE Communications Surveys Tutorials*, 2012.

[91] S. Lu, R. Ostrovsky, A. Sahai, H. Shacham, and B. Waters. Sequential aggregate signatures, multisignatures, and verifiably encrypted signatures without random oracles. *Journal of Cryptology*, 26(2), 2012.

[92] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: A tiny aggregation service for ad-hoc sensor networks. *SIGOPS Operating Systems Review*, 2002.

[93] A. Mahimkar and T. Rappaport. SecureDAV: A secure data aggregation and verification protocol for sensor networks. In *IEEE Global Telecommunications Conference*, 2004.

[94] J. McCune, E. Shi, A. Perrig, and M. Reiter. Detection of denial-of-message attacks on sensor network broadcasts. In *IEEE Symposium on Security and Privacy*, 2005.

[95] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *Proceedings of the 2010 IEEE Symposium on Security & Privacy*, S&P '10, 2010.

[96] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and A. Seshadri. How low can you go?: Recommendations for hardware-supported minimal tcb code execution. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, 2008.

[97] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. *SIGOPS Operating Systems Review*, 42(4), 2008.

[98] C. Medaglia and A. Serbanati. An overview of privacy and security issues in the Internet of Things. In *The Internet of Things*. 2010.

[99] R. C. Merkle. Protocols for public key cryptosystems. In *IEEE Symposium on Security and Privacy*, 1980.

[100] S. Nath, H. Yu, and H. Chan. Secure outsourced aggregation via one-way chains. In *ACM International Conference on Management of Data*, 2009.

[101] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. V. Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, 2013.

[102] I. D. O. Nunes, G. Dessouky, A. Ibrahim, N. Rattanavipanon, A.-R. Sadeghi, and G. Tsudik. Towards Systematic Design of Collective Remote Attestation Protocols. In *Proceedings of the 39th IEEE International Conference on Distributed Computing Systems*, ICDCS '19, 2019.

[103] Open Mobile Alliance. Lightweight Machine to Machine Technical Specification, v 1.0. Technical report, 2016.

[104] OpenSim Ltd. OMNeT++ discrete event simulator. `http://omnetpp.org/`, 2015.

[105] S. Ozdemir. Secure and reliable data aggregation for wireless sensor networks. In *Ubiquitous Computing Systems*. 2007.

[106] S. Papadopoulos, A. Kiayias, and D. Papadias. Exact in-network aggregation with integrity and confidentiality. *IEEE Transactions on Knowledge and Data Engineering*, 2012.

[107] T. Park and K. G. Shin. Soft tamper-proofing via program integrity verification in wireless sensor networks. *IEEE Transactions on Mobile Computing*, 4(3), 2005.

[108] B. Parno, J. McCune, and A. Perrig. Bootstrapping trust in commodity computers. In *IEEE Symposium on Security and Privacy*, 2010.

[109] D. Perito and G. Tsudik. Secure code update for embedded devices via proofs of secure erasure. In D. Gritzalis, B. Preneel, and M. Theoharidou, editors, *Computer Security – ESORICS 2010*, 2010.

[110] A. Perrig, R. Szewczyk, J. D. Tygar, V. Wen, and D. E. Culler. Spins: Security protocols for sensor networks. *Wireless networks*, 8(5), 2002.

[111] N. L. Petroni, Jr., T. Fraser, J. Molina, and W. A. Arbaugh. Copilot — A coprocessor-based kernel runtime integrity monitor. In *USENIX Security Symposium*, 2004.

[112] J. Rattner. Extreme scale computing. ISCA Keynote, 2012.

[113] T. Ristenpart and S. Yilek. The power of proofs-of-possession: Securing multiparty signatures against rogue-key attacks. In *Advances in Cryptology*, volume 4515 of *EUROCRYPT '07*, 2007.

[114] M. Rubenstein, C. Ahler, and R. Nagpal. Kilobot: A low cost scalable robot system for collective behaviors. In *IEEE International Conference on Robotics and Automation*, 2012.

[115] M. Rubenstein, A. Cornejo, and R. Nagpal. Programmable self-assembly in a thousand-robot swarm. *Science*, 2014.

[116] A.-R. Sadeghi and S. Schulz. Extending ipsec for efficient remote attestation. In *Proceedings of the 14th International Conference on Financial Cryptograpy and Data Security*, FC'10, 2010.

[117] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a tcg-based integrity measurement architecture. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, 2004.

[118] D. Schellekens, B. Wyseur, and B. Preneel. Remote attestation on legacy operating systems with trusted platform modules. *Science of Computer Programming*, 74(1), 2008.

[119] S. Schulz, A.-R. Sadeghi, and C. Wachsmann. Short paper: Lightweight remote attestation using physical functions. In *ACM Conference on Wireless Network Security*, 2011.

[120] A. Sehgal, V. Perelman, S. Kuryla, and J. Schonwalder. Management of resource constrained devices in the internet of things. *IEEE Communications Magazine*, 50(12), 2012.

[121] A. Seshadri. *A Software Primitive for Externally-verifiable Untampered Execution and Its Applications to Securing Computing Systems*. PhD thesis, Carnegie Mellon University, 2009. AAI3382437.

[122] A. Seshadri, M. Luk, and A. Perrig. SAKE: Software attestation for key establishment in sensor networks. In *Distributed Computing in Sensor Systems*. 2008.

[123] A. Seshadri, M. Luk, A. Perrig, L. van Doorn, and P. Khosla. Using fire & ice for detecting and recovering compromised nodes in sensor networks. Technical report, Carnegie Mellon University, 2004.

[124] A. Seshadri, M. Luk, A. Perrig, L. van Doorn, and P. Khosla. SCUBA: Secure code update by attestation in sensor networks. In *ACM Workshop on Wireless Security*, 2006.

[125] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In *ACM Symposium on Operating Systems Principles*, 2005.

[126] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. SWATT: Software-based attestation for embedded devices. In *IEEE Symposium on Security and Privacy*, 2004.

[127] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, 2007.

[128] M. Shah, S. Gala, and N. Shekokar. Lightweight authentication protocol used in wireless sensor network. In *International Conference on Circuits, Systems, Communication and Information Technology Applications*, 2014.

[129] M. Shaneck, K. Mahadevan, V. Kher, and Y. Kim. Remote software-based attestation for wireless sensors. In *Proceedings of the Second European Conference on Security and Privacy in Ad-Hoc and Sensor Networks*, ESAS'05, 2005.

[130] U. Shankar, M. Chew, and J. D. Tygar. Side Effects Are Not Sufficient to Authenticate Software. In *Proceedings of the 13th USENIX Security Symposium*, 2004.

[131] Z. Sheng, C. Mahapatra, C. Zhu, and V. C. Leung. Recent advances in industrial wireless sensor networks toward efficient management in IoT. *IEEE Access*, 3, 2015.

[132] S. Skorobogatov. Physical attacks on tamper resistance: Progress and lessons. 2011.

[133] S. Skorobogatov. Physical attacks and tamper resistance. In *Introduction to Hardware Security and Trust*. 2012.

[134] S. P. Skorobogatov. *Semi-invasive attacks: a new approach to hardware security analysis*. PhD thesis, Citeseer, 2005.

[135] G. Spanogiannopoulos, N. Vlajic, and D. Stevanovic. A simulation-based performance analysis of various multipath routing techniques in ZigBee sensor networks. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. 2010.

[136] D. Spinellis. Reflection as a mechanism for software integrity verification. *ACM Transactions on Information and System Security (TISSEC)*, 3(1), 2000.

[137] P. Stelling, C. DeMatteis, I. T. Foster, C. Kesselman, C. A. Lee, and G. von Laszewski. A fault detection service for wide area distributed computations. *Cluster Computing*, 2(2), 1999.

[138] R. Strackx, F. Piessens, and B. Preneel. Efficient isolation of trusted subsystems in embedded systems. In S. Jajodia and J. Zhou, editors, *Security and Privacy in Communication Networks*, 2010.

[139] E. Syta, I. Tamas, D. Visher, D. I. Wolinsky, L. Gasser, N. Gailly, and B. Ford. Keeping authorities"honest or bust"with decentralized witness cosigning. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, S&P '15, 2015.

[140] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, S&P'13, 2013.

[141] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, S&P '15, 2013.

[142] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, 2006.

[143] Trusted Computing Group. *PC client specific TPM interface specification (TIS)*, 2004. Version 1.2, Revision 1.00.

[144] Trusted Computing Group (TCG). Website. http://www.trustedcomputinggroup.org, 2015.

[145] T. Unterluggauer and E. Wenger. Efficient pairings and ECC for embedded systems. In *Cryptographic Hardware and Embedded Systems*, CHES '14, 2014.

[146] A. Valmari. The state explosion problem. In *Lectures on Petri nets I: Basic models*. 1998.

[147] R. van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, Middleware '98, 1998.

[148] J. Vijayan. Stuxnet renews power grid security concerns, 2010.

[149] J. Vijayan. Target attack shows danger of remotely accessible HVAC systems. `http://www.computerworld.com/article/2487452/cybercrime-hacking/target-attack-shows-danger-of-remotely-accessible-hvac-systems.html`, 2014.

[150] J. won Ho. Distributed detection of node capture attacks in wireless sensor networks. 2010.

[151] G. Wurster, P. C. Van Oorschot, and A. Somayaji. A generic attack on checksumming-based software tamper resistance. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, S&P '05, 2005.

[152] Y. Yang, X. Wang, S. Zhu, and G. Cao. Distributed software-based attestation for node compromise detection in sensor networks. In *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems*, SRDS '07, 2007.

[153] Y. Yang, X. Wang, S. Zhu, and G. Cao. Sdap: A secure hop-by-hop data aggregation protocol for sensor networks. *ACM Transactions on Information and System Security*, 11(4), 2008.

[154] Z. Yu and Y. Guan. A key management scheme using deployment knowledge for wireless sensor networks. *IEEE Transactions on Parallel and Distributed Systems*, 2008.

[155] S. Zeitouni, G. Dessouky, O. Arias, D. Sullivan, A. Ibrahim, Y. Jin, and A.-R. Sadeghi. Atrium: Runtime attestation resilient under memory attacks. In *2017 International Conference On Computer Aided Design (ICCAD'17)*, 2017.

[156] W. Zhang, Y. Liu, S. K. Das, and P. De. Secure data aggregation in wireless sensor networks: A watermark based authentication supportive approach. *Pervasive and Mobile Computing*, 2008.

[157] J. Zhao and R. Govindan. Understanding packet delivery performance in dense wireless sensor networks. In *International Conference on Embedded Networked Sensor Systems*, 2003.

[158] K. Zhao and L. Ge. A survey on the Internet of Things security. In *International Conference on Computational Intelligence and Security*, 2013.

[159] C. Zhong, Y. Mo, J. Zhao, C. Lin, and X. Lu. Secure clustering and reliable multi-path route discovering in wireless sensor networks. In *Symposium on Parallel Architectures, Algorithms and Programming*, 2014.

[160] Y. Zhou and D. Feng. Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing. *IACR Cryptology ePrint Archive*, 2005, 2005.

Erklärung gemäß §9 der Promotionsordnung

Hiermit versichere ich, die vorliegende Dissertation selbstständig und nur unter Verwednung der angegebenen Quellen und Hilfsmittel verfasst zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, Germany, March 2019

Ahmad Ibrahim